The BaBar Calibration System

David Brown, LBL
Alex Romosan, LBL
Vasili Shelkov, LBL
Todd Stavish, LBL
July 9, 1998

Draft 3.3

Abstract

The BaBar Calibration System is a toolkit of classes designed to facilitate the implementation of calibration. It includes tools for database storage and retrieval of calibration data, for measuring calibration constants, and for interactive examination of calibration results. This toolkit is supplemented by infrastructure classes designed to support it in both the online and offline programming environments. This document describes both the programming API and the underlying support software.

Figures

Contents

1 Introduction

The BaBar calibration system is designed to facilitate the calibration of the BaBar detector. This system has been adopted for electronics calibration by the BaBar online group and for performing datastream calibrations in Prompt Reconstruction by the BaBar reconstruction group. In this document, we describe the design and implementation of this system.

The BaBar calibration system provides specific solutions for dealing with many of the technical aspects of the specific coding environments found in BaBar (dataflow, OEP, and reconstruction). This minimizes the amount of expert knowledge necessary for subsystem developers to create calibration applications. It also presents a uniform interface to calibration users across all subsystems, allowing code reuse between subsystems.

The BaBar calibration system is not a framework like dataflow, OEP, or the offline reconstruction and analysis frameworks. It is instead a toolkit, which provides developers with standard interfaces and implementations. It contains tested code, which can be used to reduce the developer coding burden. This toolkit takes the form of a set of C++ classes which work together to provide the needed functionality.

This document will be kept current as the packages develop. The most recent version can always be found at the following URL: http://www.slac.stanford.edu/BFROOT/doc/www/Computing/Online/OnlineCalibration.html. The reader is assumed to be familiar with the BaBar online and offline systems, and to have a basic understanding of C++ and OO software. This document is targeted mainly for calibration developers, but calibration consumers may also find it useful.

2 Design Overview

The calibration system design is based on a particular definition of calibration. We define calibration to be the characterization of the response of the detector and/or detector electronics to known stimuli. The purpose of calibration is to record those responses, in order to provide a better understanding of the physics significance of the event data, and to allow optimization of the detector for taking subsequent event data.

In calibration, the detector response is observed through the same electronics used for physics data. The stimulus may be artificial (such as charge injection) or natural (a track). Response characterization takes the form of determining the parameters of a function family (defined a-priori) which best describe the response in terms of the known stimulus. Note that, by this definition, alignment is not a calibration, since that characterizes the position of the active elements of the detector, not the direct electronic response. Similarly, measurements made using ancillary hardware readout independently of the detector electronics (magnet current, atmospheric pressure, beam orbit monitors, etc.) are not calibrations. A characterizable detector response dependence on those measurements however would be a legitimate calibration (for instance, drift time dependence on atmospheric pressure).

The end result of a calibration may be simply the function parameters. It may also be an optimal value of the function ordinate given the measured response. For instance, the result of a threshold calibration might be the threshold value that maximized the signal/noise, whose value as a function of threshold was parameterized in calibration.

In order to characterize fully a system's response, many different calibration measurements are required. For example, to characterize a typical ADC channel both its gain and pedestal must be measured. Different measurements made on the same channel are referred to as calibration types. The term channel can refer to any unit of the electronic readout hierarchy below the readout module (ROM), such as front-end channels, chips, sections, or modules. Subsystems must design their own calibration types according to their needs, using the base classes and templates provided by the core software.

The BaBar calibration system divides the process of performing a calibration into well-defined steps. To allow an accurate response characterization, the stimuli must be repeated to average out statistical fluctuations. This loop over stimuli is referred to as pulsing. For each stimulus in the pulsing loop, the measured response is used to update the state of some object recording the response for the entire loop. This process is referred to as accumulation. Once sufficient stimuli have been accumulated, this object is processed to extract the parameters. This is referred to as fitting. Final parameters can be either verified (compared against a reference, and judged compatible or incompatible), or validated (judged accurate or inaccurate). Once validated, parameters can be stored in the calibration database. The software for supporting these different phases of performing a calibration is described in detail in the rest of this document.

An important concern that shaped this design was assuring data safety and quality. Calibration data is crucial to the interpretation of the event data and is irreplaceable if damaged or destroyed. This concern is especially prominent when using Objectivity, whose native API provides read-write-modify access to all users. This opens the possibility for inadvertent modification or destruction of calibration data by user code. To guard against data damage, users interact with the calibration system only through transient copies of persistent objects. The calibration system also provides safeguards on the quality of the stored data through gatekeeper objects that test the structure and content of calibration data before allowing it to be persistently stored. Together these strategies provide users with full read and write access to the calibration database without risking data quality or integrity.

Calibration data are stored according to the time-interval structure defined by the Conditions Database. This defines a time validity range for all constants, with the most-recently stored set generally assigned an open-ended range (valid until time +infinity). Providing alternative versions of data for the same time period is supported by the conditions database. Details of the conditions database are described in a separate document

Another important aspect of the calibration system design is its interface to the BaBar online system. In order to exploit the direct access to raw data and the processing power of the readout module (ROM) farm, the accumulation and fitting stages of the calibration were designed to be able to run in the ROM. To facilitate debugging and reuse of that code, the calibration classes for accumulation and fitting also function under Unix. To minimize the physics luminosity cost of running online calibration, this code must be both efficient and robust. These requirements have been incorporated into the design of the calibration system.

3 Calibration Packages

The calibration code is divided into several packages that follow the general guidelines for online, offline, and dual-use packages. **BbrCalib** is a dual-use package that contains most of the base classes used throughout the calibration system. **CalDatabase** is an offline package which implements the calibration interface with the conditions database, and defines the persistent base classes for storing calibration data. **CalOnline** is a dual-use package which defines classes used in both the Unix and dataflow parts of the calibration system (tagged containers and iterators). **CalOdf** is an online-only package that defines the calibration interface to dataflow. **CalFit** is a dual-use package that provides statistical tools for fitting accumulated data. **CalOffline** is an offline-only package that contains tools for offline manipulation and visualization of calibration data. **BdbBrowser** is an offline-only package with a GUI for browsing calibration data.

Subsystems are expected to manage several packages for their own calibration-related software. We suggest a common naming scheme for these packages, in order to make it easier for non-subsystem experts to find code. Examples of these packages now exist for most of the subsystems, and can be found by looking in a recent offline release. The ***Xxx*Cond** package should define the persistent classes used in calibration. Here, *Xxx* stands for the three-letter acronym of a subsystem. **XxxCond** is an offline-only package. Subsystem classes for accumulating and fitting online calibration may be kept in the ***Xxx*Calib** or ***Xxx*Online** packages. These should be dual-use packages, as accumulation and fitting should be transportable between dataflow and OEP. Subsystem classes for datastream calibrations may be kept in the ***Xxx*OEP** or ***Xxx*PromptReco** package. ***Xxx*Env** packages define the reconstruction interface to the conditions database, and should provide appropriate proxies for calibration persistent data. Other kinds of conditions database classes (IE geometry and alignment) also have proxies in these packages.

4 The Basic Calibration Classes

The architecture of the calibration system is defined by several classes in the **BbrCalib** package. This package contains base classes for all of the transient (non-Objectivity) functionality of the calibration system. It also contains some fully implemented generic classes that can be used as-is for simple calibration types (pedestals, gains, etc.), and serve as examples for subsystem specific implementations. These Classes are described below.

4.1 CalChan

CalChan is the base class for accumulating and storing calibration constants for a single channel. It is the central class in the calibration system, being referenced by nearly every other class, and used in all phases of the calibration process in both the online and offline environments. The base class is simple, yet supports subclassing to describe elaborate calibrations. CalChan is not a persistent capable class. Instead, it is stored persistently through inclusion in a persistent class.

The CalChan base class contains a channel ID that uniquely defines the channel. The interpretation of a channel ID is context-dependent. When accessed during verification or validation, or when stored in Objectivity, channel ID generally refers to a valid dataflow detector tag. When used within the dataflow system, the ID generally refers to the detector address. The only exceptions to these interpretations of the CalChan channelID are when indirection is used to compress redundant data. Indirections are discussed in detail in section 12.

The CalChan base class also provides a channel flag, which is used to summarize the status, condition, and properties of its data. The flag is divided into private fields, whose allowed values are predefined and whose setting is controlled, and public fields, which can be separately defined for each calibration type. The CalChan flag fields are described in detail in section 15.1.

Because of Objectivity restrictions when storing collections of objects, and because of dataflow transmission restrictions, CalChan subclasses can only contain fixed-length collections of fundamental data types (IE they cannot contain pointers, strings, lists, etc.). To insure platform independence and compatibility with Objectivity, the fundamental types used in CalChan and its daughters must be defined using the typedefs found in the file BaBar/BaBarODMBTypes.h.

CalChan provides a virtual interface to access both the base class and subclass data members, which is used in many parts of the calibration system. For instance, the name of every CalChan subclass is through the ‘ channelName’ function. In addition to the explicitly required pure virtual functions, CalChan subclasses are implicitly required to provide functional default constructors, copy constructors, and equivalence operators. Absence of these will cause template instantiation failure in user code. In addition to the default constructor, it is recommended that all CalChan subclasses provide a constructor which takes as input the channelID, and which initializes all specific data members to default values.

The ‘channel’ a CalChan object represents is not necessarily a single electronics channel. For instance, a single CalChan object can describe a threshold value common to all channels in a chip. A CalChan is intrinsically capable of being associated with any of the dataflow detector levels (module, section, chip, or channel). Association of a single CalChan object to arbitrary collections of electronics channels can be via an indirection. In all cases, the significance of what level or collection a CalChan object represents is not stored in the CalChan itself, but is provided by the calibration type (see section 4.1.4). Thus the same CalChan subclass may be used to represent channels in one type and chips in another.

CalChan subclasses are intended to be simple classes providing only basic data storage and access. Thus a single CalChan subclass can be used to store the constants from different calibration types if their data format (float, int, etc.) are identical. As with the channelID, the interpretation of CalChan data fields is provided by the calibration type. Thus the same CalChan subclass may be used in one calibration type to represent ADC values and in another TDC values.

BbrCalib contains several generic implementations of CalChan. A UML diagram of these is shown in Figure 1, and they are described in detail below.

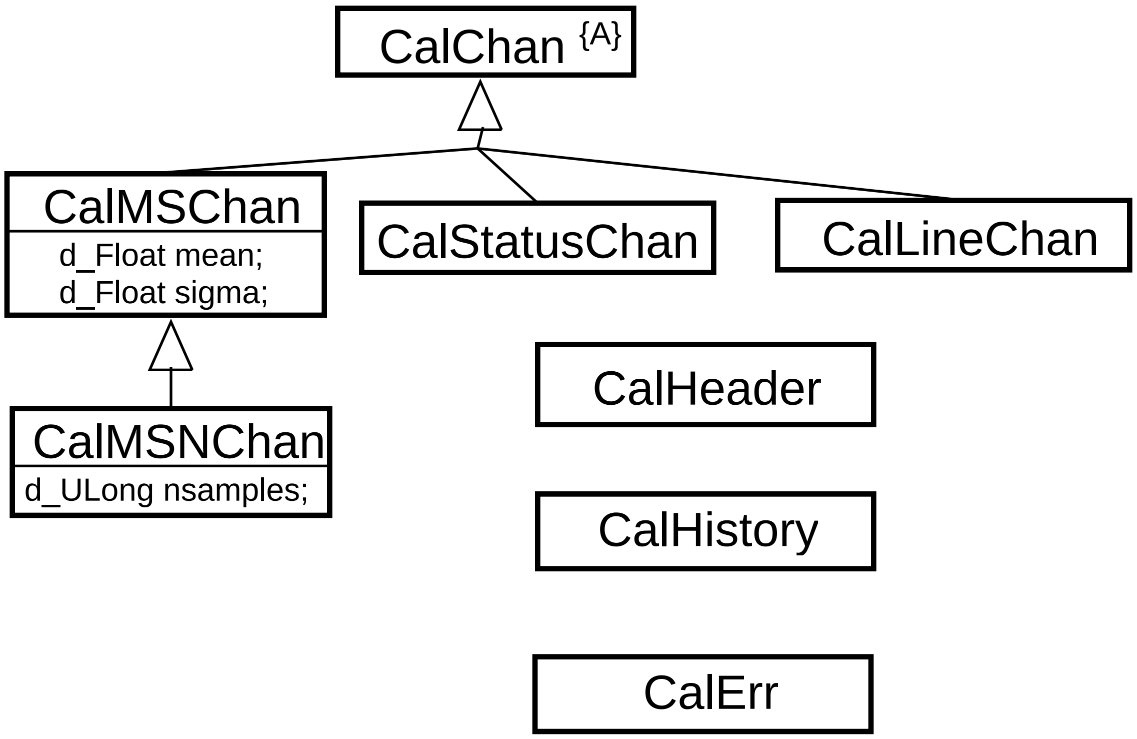

Figure 1 Generic Calibration Classes

4.1.1 CalStatusChan

CalStatusChan is the simplest possible implementation of a CalChan in that it has no data members besides those defined by the base class. This class is intended to represent the general status for a channel, and can be used for the required *Xxx*StatusType calibration type. Private and public fields of the normal CalChan flag operate exactly the same for this class as for any other, except that they are interpreted as the general condition of the channel, not just how it performed in a single calibration type.

The responsibility for merging information from various calibration types into the CalStatusChan general flag lies with the subsystem developer. The code for performing the merger should be part of the *Xxx*StatusType class. Since this type will be used throughout the online and offline, it probably should be updated only rarely, in order to provide stability for users.

4.1.2 CalMS[N]Chan

Two very similar generic CalChan subclasses (CalMSChan and CalMSNChan) are implemented for storing simple calibration data. These classes store a mean value and RMS as floats. CalMS[N]Chan objects can be used to represent any value whose sampling obeys Gaussian statistics: pedestals and gains are obvious examples. The only difference between these classes is in how they handle sample counting: CalMSChan assumes an external counter has kept track of the number of samples used to compute the value and RMS, while CalMSNChan stores the number of samples itself. CalMSChan is only appropriate when exactly the same number of samples will be used to compute values for all the channels in a CalBank, in which case the number of samples should be stored by the CalHeader object. CalMSNChan should be used whenever the number of samples may vary by channel. CalMSNChan is thus more general, while CalMSChan is more efficient in using storage space. Example use cases of CalMS[N]Chan classes are given in section 6.

4.2 CalHeader

Information common to a collection of CalChan objects is stored in a CalHeader object. CalHeader is a fully implemented class at base. If additional common information is needed to interpret a CalChan collection, CalHeader should be subclasses and that information should be stored in the CalHeader subclass. This guarantees that the information will not be lost in processing, transporting or storing the collection.

The CalHeader base class stores the number of pulses processed to create the data. It also defines a flag that describes the general condition and status of the collection. For instance, the reference bin in the CalHeader flag identifies banks used as a standard against which to compare new data during verification or validation. Flag selection will also be used to create a functional subset of the calibration database for efficient export to remote sites based on the nature of the intended remote usage, though this feature is not yet implemented. A full description of the CalHeader flags is given in section15.2.

4.3 CalErr

CalErr is a lightweight class copied directly from the TrkErr class used in tracking. It defines success and failure codes and associates them with text strings. The CalErr functions ‘success’ and ‘failure’ return Boolean values which summarize the code value. Code values below 10 have predefined text strings; values above 10 can be associated with an arbitrary text string. Many functions that interact with the database return their status via a CalErr object. Users are strongly advised to always test the return value of any function returning a CalErr. CalErr objects can also be printed using the ostream << operator.

4.4 CalHistory

The CalHistory class is used to record changes made on calibration data. When modifications are made to calibration data a CalHistory object is automatically created. During a CalHistory object's creation, it logs information regarding the change to the calibration data and information about the user who initiated the change. By keeping collections of CalHistory objects, a journal is produced that accurately tracks all modifications to calibration data. This journal can be used both to keep track of interactive operations when examining and manipulating data inside transient objects, and for recording the set of manipulations used to create persistent data.

In the CalHistory class, five data fields are used to register the necessary information concerning alterations to calibration data. These fields are:

- A description of the alteration
- The time the alteration occurred
- The UNIX user name and ID of the person performing the alteration
- An index that is used when merging CalHistory collections.

CalHistory is a non-persistent capable class composed of fundamental data types to allow it to be persistently stored in an ooVArray. CalHistory objects are data members of the CalBase, CalBank, and CalType classes.

4.5 CalSimpleMapBrowser

A common use case in calibration involves creating an object with a complete set of channels whose channelIDs correspond to some part (say a ROM) of the hardware platform. For instance, this need arises in initializing storage for accumulation or fitting. The channelID spaces relevant for calibration are most fully defined by the 'map' classes in the MapDetector package1 . In order to decouple the specific implementation of these maps from the calibration code, the MapAbsDetBrowser class in the MapOnline package is used in calibration.

MapAbsDetBrowser functions as an iterator whose 'next' function returns sequentially all the individual channelIDs of the objects between two layers of the dataflow system. For instance, a MapAbsDetBrowser may be instantiated to browse over channels (lower level) specific to a particular ROM (upper level). Calling 'reset' on a MapAbsDetBrowser object returns it to the state it had on construction, allowing it to be reused

The CalSimpleMapBrowser class is provided in the BbrCalib package as a simple implementation of MapAbsDetBrowser, appropriate only for debugging of systems with a simple platform hierarchy. More sophisticated implementations of MapAbsDetBrowser are available in the MapDetector and MapOdf packages. CalSimpleMapBrowser works only in detector tag space, and assumes a compact, uniform channelID space. This class actually contains its 'map' in specific data members whose value are defined on construction.

Implementations of CalMapBrowser based on the MapDetector classes will be provided in a future release of the calibration code.

5 Accumulating Calibration Data

CalAddChan is an abstract CalChan subclass that adds special functions associated with accumulation. These functions allow the CalAddChan object state to be updated for new data presented on each pulse, or to prepare it for a new sequence of pulses.

The primary CalAddChan accumulation function is ‘increment’, which passes pulse data to the object. This function takes as input an object of class AbsArg1, which functions as a type-safe void . With this prototype, increment can be implemented in the CalAddChan subclasses to take whatever type (IE float, int, or more complicated structures) is appropriate for that subclass. The cast from an AbsArg to the correct type is done via the AbsArgCast function, which insures type safety at run time. For instance, if one tries to cast an AbsArg representing a float to an int, the result will be a null pointer, whereas casting it to a float gives a pointer to a float. AbsArg is a very lightweight class that does not appreciably impact the performance of the increment method.

As with all CalChan subclasses, CalAddChan subclasses are required to have only simple (fixed size) data members, to allow them to be stored in the database. Storing a bank of CalAddChans may be useful during commissioning and debugging. During factory-mode running the CalAddChans will probably not be stored, but instead the CalChan output to the fit of the CalAddChan will be stored.

The **BbrCalib** package contains several fully implemented CalAddChan subclasses. These are intended to be used directly by the subsystems, as well as serving as examples for subsystem specific accumulation implementations. These classes cover many of the accumulation use cases described by the subsystems in their responses to the calibration survey. A UML diagram of these classes is shown in Figure 2.

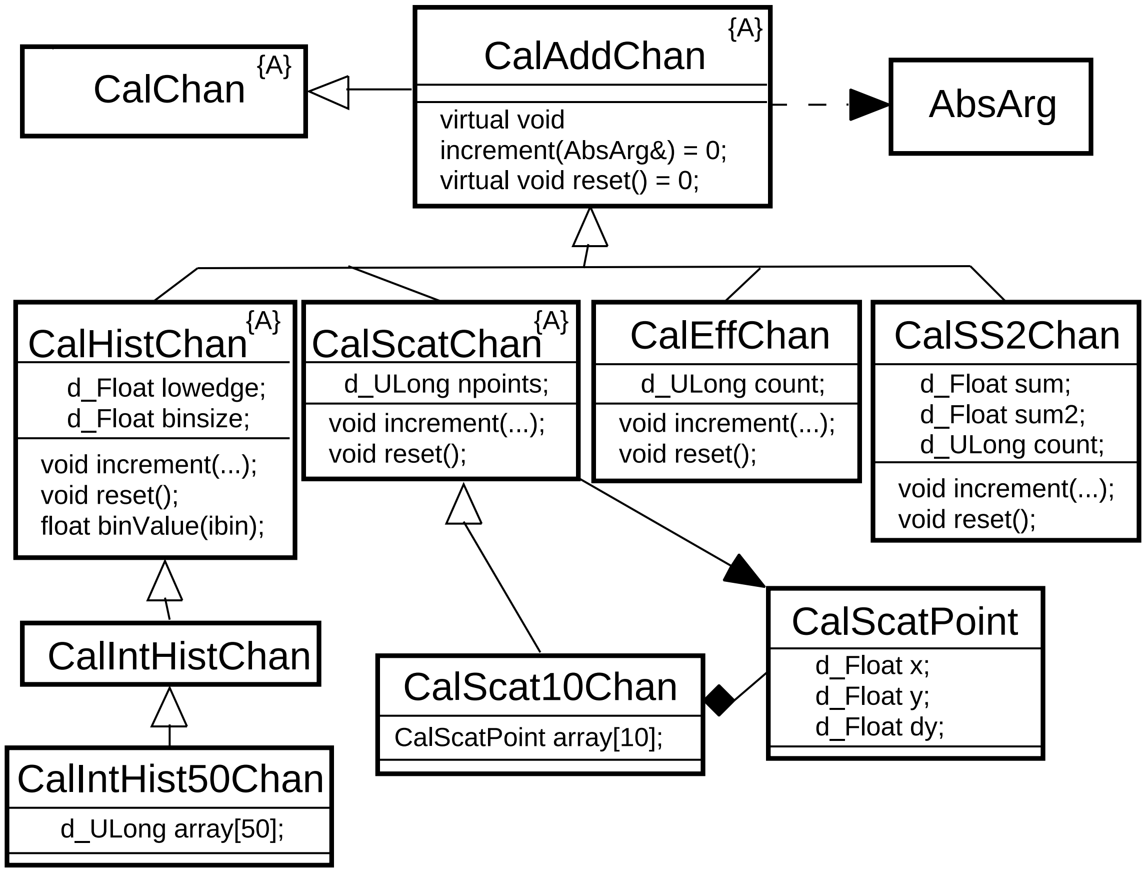

Figure 2 CalAddChan and Subclasses

5.1 CalEffChan

The simplest CalAddChan is the CalEffChan, which is intended for measuring hit efficiency. This class increments a counter every time increment is called, regardless of the value of the AbsArg. A similar CalAddChan subclass is CalSS2Chan, which keeps a running sum, sum of squares, and counter. CalSS2Chan can use either int or float AbsArg inputs. The CalFit class that processes these channels into useful quantities (like efficiencies or means and sigmas) is described in the next section.

5.2 CalHistChan

A more complicated CalAddChan is CalHistChan. This base class defines an interface for interpreting channel data as a histogram. In CalHist, the input argument passed to the increment function is a float, which is used to increment an appropriate bin. The CalHistChan base class stores all information necessary to interpret the histogram (bin range and size, etc) and provides the usual set of histogram access functions (bin contents and center position by index, etc). It does not however define the type or storage of the bin contents. A particular CalHist subclass CalIntHistChan defines bin storage as 32-bit integers, with appropriate definition of the access functions. An implementation which allows weighted histograms is possible under the CalHist interface, but not yet implemented.

Because CalChan data members must be fixed-size, CalIntHistChan cannot be fully implemented for an arbitrary number of bins. Instead it must be subclassed to provide bin contents storage as a fixed size array. The class CalIntHist50Chan is provided as a 50-bin example of a fully implemented CalHistChan subclass.

In order to inspect CalHistChan contents, a translation to a viewable histogram form is needed. This is provided by the CalHistBook class described in section 6, which translates CalHistChans into Heptuple histograms.

5.3 CalScatChan

Another CalAddChan subclass implemented in **BbrCalib** is CalScatChan. This class stores a true scatterplot of points defined by the CalScatPoint class, which stores *x*,*y*, and *dy* as floats. The CalScatChan implementation of increment requires the AbsArg to be of type CalScatPoint and simply copies this into the CalScatChan's local store. As with CalHistChan, an arbitrary number of points cannot be stored in CalScatChan, hence it must be subclassed to provide fixed-size storage. The CalScat10Chan class is provided as a 10-point example.

CalScatChan is intended to be used in nested cycle calibrations, where the result of each low-level cycle is entered as a point in a scatter plot. A simple example is provided in the **CalOnline** test program TestCalFit.

6 Fitting Accumulated Data

After accumulating statistics, it is generally necessary to post-process or finalize the data to produce calibration constants. This process is referred to as fitting. The calibration system uses the CalFit class to describe the fit interface. A UML diagram of CalFit and its relationship with other calibration classes is shown in Figure 3.

CalFit is a templated abstract base class from which all calibration fits inherit. The term fit refers to the final processing of a CalAddChan after it's been incremented for the last time. CalFit processing does not change the internal state of the CalAddChan, but instead produces a new CalChan object to store the processing result. Storing the processing result as a separate object guarantees that the intermediate calibration results (the CalAddChan objects) can be brought up through dataflow in the same state as when they were processed, to allow debugging of the CalFit object offline.

The CalFit base class defines the function ‘fitChannel’, which takes an input CalAddChan and sets the state of an output CalChan whose subclass is specified by the CalFit template argument. The caller provides the output channel object as an argument to this function. To be used online, the fitChannel function must be implemented as a deterministic and robust algorithm, as its call will not be protected. The output of the fitChannel function is a FitStatus flag, whose value can signal any irregular conditions that arise during the fit. This return value should also be encoded into the FitStatus field of the output CalChan object, allowing tests on the fit status to be deferred to verification or validation if desired.

The **CalFit** package contains generic and specific implementations of CalFit subclasses. The more sophisticated implementations provided in **CalFit** have been developed by Matt Weaver of Caltech, and are described in dedicated documentation which can be found at the following URL: http://www.slac.stanford.edu/~weaver/CalFit/intro.html

The **CalFit** package contains three fully implemented CalFit subclasses, which serve as simple examples for developing subsystem-specific CalFit subclasses, as well as for direct use in simple calibrations. CalSS2MSNFit has a fitChannel function which converts a CalSS2Chan (which stores Σx, Σx2, and Σ1) to a CalMSNChan (which stores mean, sigma, and N). The ‘fit’ in this case consists only of simple arithmetic operations.

The CalHistLineFit and CalScatLineFit classes provide a more sophisticated CalFit example. These classes perform a closed-form least squares linear fit to a CalHistChan or a CalScatChan, respectively. The minimization engine used for this fit is an object of class CalLeastSquares, an independent class which performs a closed-form minimization of any function when supplied with residuals and derivatives. CalLeastSquares provides a full-featured and general interface to allow it to be used by other CalFit subclasses that need minimization engines. Since the algorithm used is closed form and all ‘risky’ operations (such as matrix inversion) are internally tested, it satisfies the CalFit online use requirements.

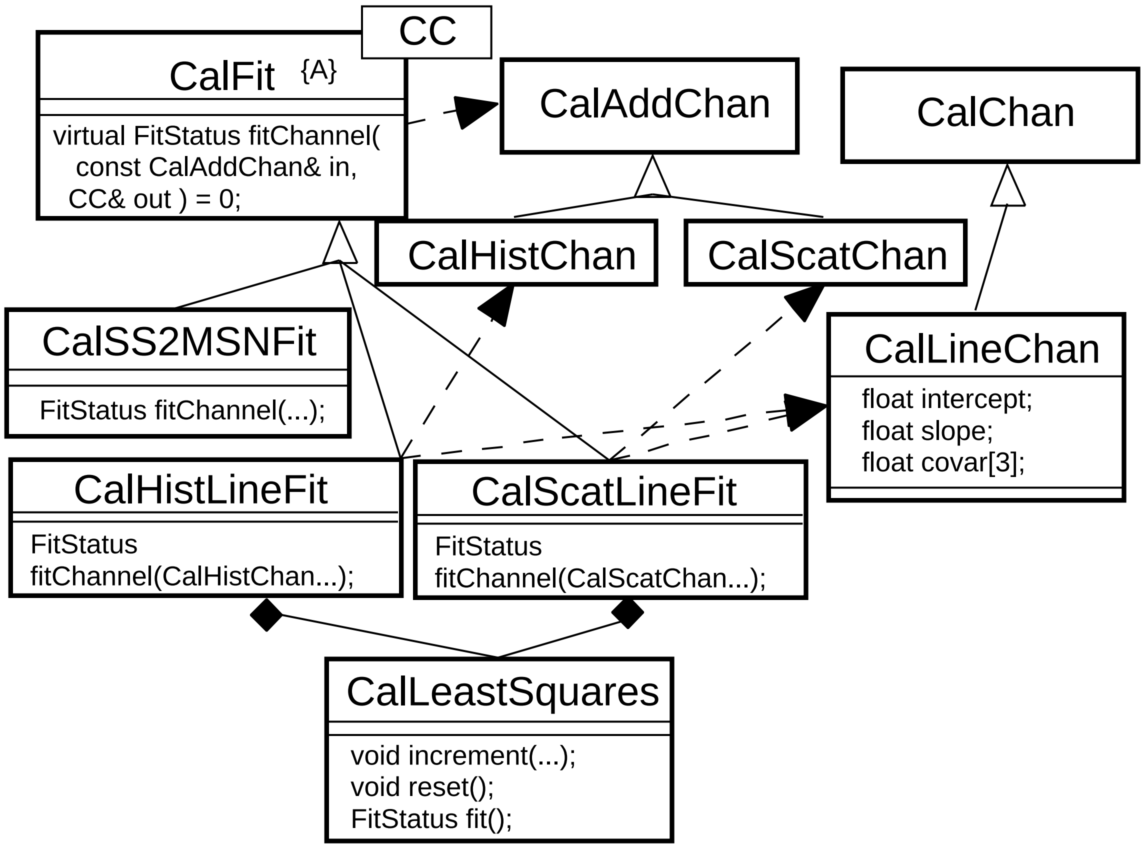

Figure 3 CalFit and Related Classes

7 Processing Calibration Data

The calibration system has been designed to operate within all three of the online environments, namely dataflow, OEP, and Prompt Reconstruction (PR). However, these environments present substantially different methods and classes for accessing the different transitions and the data associated with them. Because of this, the direct interface to most calibration classes has been left very general. In particular, none of the accumulation classes mentioned above provides functions for looping over channels or data accumulation cycles. Implicitly, their interface assumes those loops will be implemented in user code, independently for dataflow, OEP, and PR

BbrCalib includes a number of classes that provide a more general interface that implements some of these loops, and can be transported freely between the online environments. These classes do not provide any new functionality, but rather are *wrapper* classes, with subclasses built around the specific collection and accumulation classes already described. They provide access to the specific functions of those specialized classes through a standardized interface.

A UML diagram of these classes is shown in Figure 4, and they are described in detail below.

Calibration system users are not required use these wrapper classes to implement their calibration accumulation. Doing so may however reduce their coding burden and result in simpler, more easily maintained code. Using these classes will also insure that their code can be moved between dataflow and OEP with relatively little work. These wrapper classes however may not provide the most efficient implementation for all types of calibration, and users are advised to judge for themselves the costs and benefits.

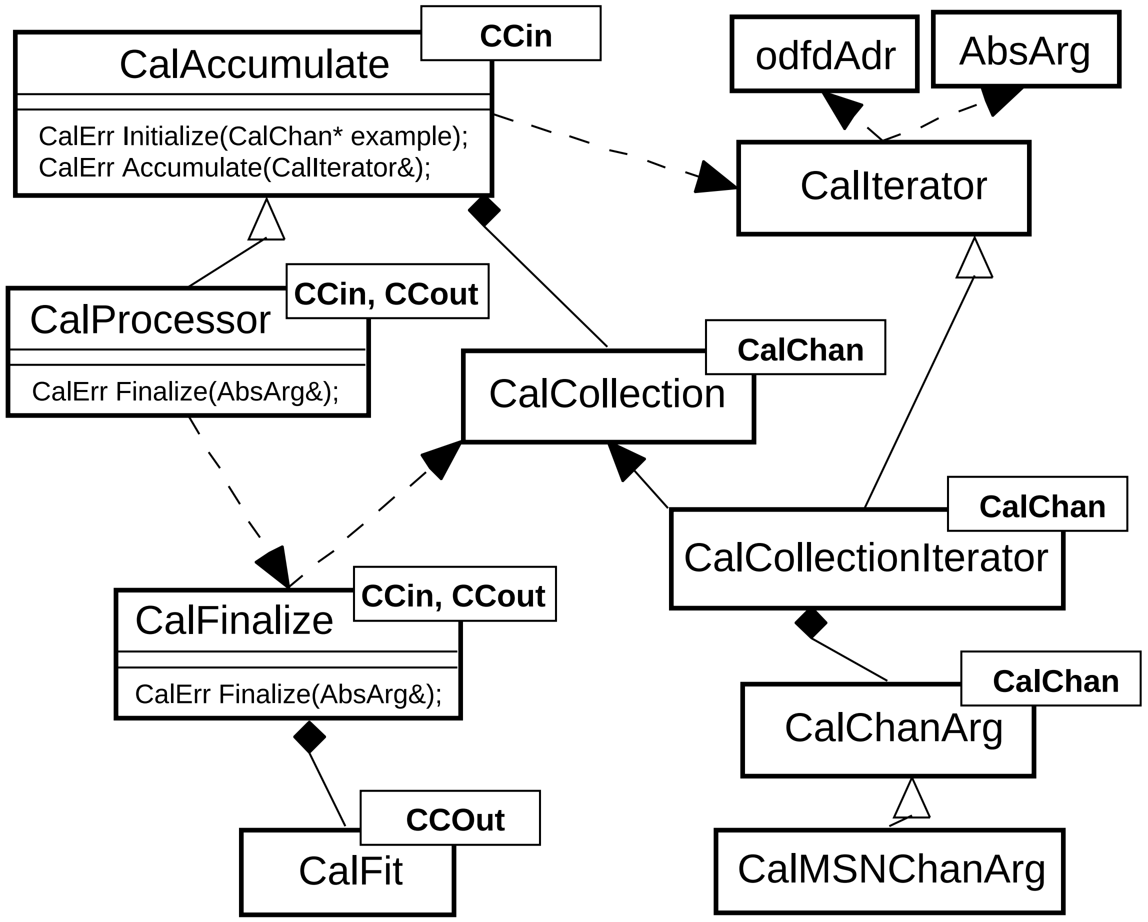

Figure 4 Calibration Classes used to Process Data

7.1 CalCollection

The container classes used to hold CalChans are specialized for different use cases in Unix (interfaces with the database) and dataflow (interfaces to the data transport system). In order for accumulation and fitting code to work in both computing environments, a generic way of handling CalChan collections is needed. The CalCollection class provides this portable interface. CalCollection is an abstract base class that defines an interface for accessing channel data by index. In addition, the ‘channelIndex’ function finds the CalChan in the collection whose channelID matches the input value. This function is implemented in the base class using a binary tree search. Specific CalCollection subclasses may wish to override this with a hash table or similar faster algorithm if the structure of their collection is known a-priori.

CalCollection is a wrapper class, which does not own its data, but can access it in the class it wraps. CalListCollection is a CalCollection subclass built around a CalChanList. A concrete implementation of CalCollection around a CalTC is provided in the **CalOnline** package. A concrete implementation around a CalChanList is provided in the **CalDatabase** package. Use cases for CalCollection objects are described in sections 7.4 and 7.5, and a UML diagram of CalCollection and subclasses is shown in Figure 5. The data classes which CalCollection subclasses wrap are described in sections 10.3 and 8.1.1 for UNIX and Dataflow uses respectively.

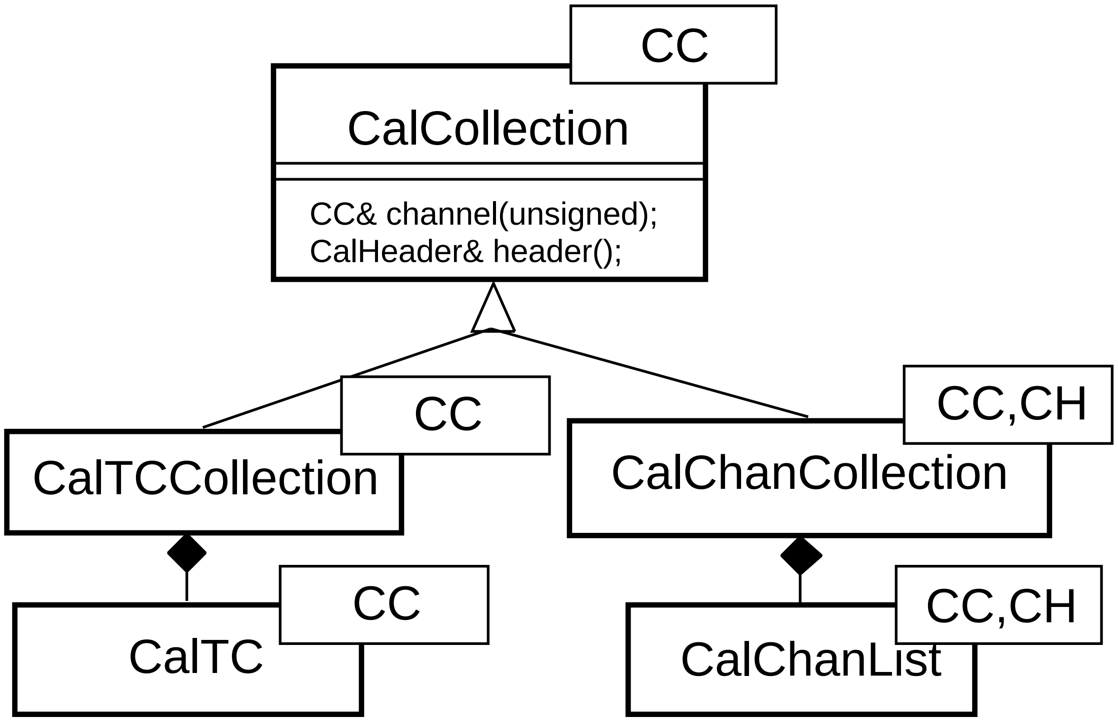

Figure 5 CalCollection and Subclasses

7.2 CalIterator

CalIterator is an abstract class that defines an interface for looping over channel-structured data and sequentially processing data from every channel into an

AbsArg. CalProcessor uses CalIterator as a standard way of accessing arbitrary data serving as input to calibration accumulation. To use CalProcessor to perform a particular kind of calibration accumulation, an appropriate subclass of CalIterator must be written. For instance, a CalIterator subclass designed to loop over the channels in the input container returned by a L1 accept of a given subsystem would be a natural way of implementing accumulation in the ROM.

The CalIterator interface is mostly defined by the 'next' function, which returns an AbsArg for every channel in the collection it is iterating over. The 'next' function also updates the value of the channelID argument (supplied by reference) to the value corresponding to the channel used to create the AbsArg. As with most other parts of the calibration system, the channelID return value must correspond to a valid detector address (in dataflow) or detector tag (in OEP or offline). The 'end' function signals the last channel in the channel-structured data when it returns true. The CalIterator 'reset' function should be implemented to return the iterator to its initial state.

Besides simply iterating over the collection, CalIterator must process the data it finds for each channel into the form expected by a particular CalAddChan. This processing function may be as simple as wrapping an AbsArg around a native data member in the channel-structured data, or it may involve calling elaborate mathematical transform functions. It is very likely that several CalIterator subclasses will be needed to loop over the same channel-structured data class when used to accumulate different calibration types. An example CalIterator implementation for 'raw' data in the DIRC is described in section 15.5.

7.2.1 CalCollectionIterator

CalCollectionIterator is a subclass of CalIterator intended for use in nested calibration cycles. It is a templated concrete class. Its constructor requires a CalCollection object and a CalChanArg object (described below), both templated on the same CalChan subclass as CalCollectionIterator itself. CalCollectionIterator implements the CalIterator functionality by simply looping over the index values of its underlying CalCollection. Its 'next' function returns the AbsArg value returned by its CalChanArg when applied to the current CalChan object. It updates the input channelID to have the same value as the current CalChan's channelID.

7.3 CalChanArg

CalChanArg is a templated abstract base class intended for use in nested calibration cycles. It defines the interface for a factory object that translates a (const) input CalChan object into an AbsArg value suitable for using in the increment method of an (unspecified) CalAddChan.

CalMSNChanArg is a fully implemented subclass of CalChanArg templated on CalMSNChan provided in CalOnline as an example. It combines the CalMSNChan mean and error together with a floating point value supplied on construction to build a CalScatPoint. The floating value is used as the CalScatPoint x value, while the mean and error are used as y and dy. The CalScatPoint owned by CalMSNChanArg is returned in the form of an AbsArg by the CalChanArg function. This class is used in one of the test programs described below.

7.4 CalAccumulate

The CalAccumulate class provides the high-level calibration processing interface. CalAccumulate is a fully implemented singly templated class that can initialize and accumulate all the channels in a CalCollection. The CalAccumulate constructor takes as input a CalCollection object for storing the accumulation intermediate result. This object us used directly by CalAccumulate (IE it is not copied), but ownership is kept outside the class. It is therefore important not to alter the objects provided to the CalAccumulate constructor, or to delete them or allow them to fall out of the CalAccumulate object scope. The same warning applies to the CalCollection classes used by CalAccumulate regarding the CalTC or CalChanList objects that they wrap.

The CalAccumulate ‘initialize’ function resets the state of all the CalAddChans in its input CalCollection. By default this is done by calling the CalAddChan ‘reset’ function of those channels. If the optional ‘example’ argument is provided, initialization is performed by equivalencing all the input channels to that example.

The CalAccumulate ‘accumulate’ function must be called by the user for each step in the accumulation loop. This function takes as input an object of class CalIterator, which is described in section 7.2. CalAccumulate implements the ‘accumulate’ function by looping over the input data specified by the CalIterator object, and calling the ‘increment’ function of the corresponding CalAddChan for each element returned by CalIterator. Thus, CalAccumulate can process entire collections of CalChans in a single function call.

CalAccumulate inherits from an untemplated base class CalAccumulateBase. CalAccumulateBase is a pure interface, defining the functions implemented by CalAccumulate. Because it is untemplated, objects of class CalAccumulate can be simply passed to other classes, allowing accumulation to be performed int those classes without their having to know what CalAddChan subclass is being accumulated.

7.5 CalFinalize

CalFinalize is a class that can fit a collection of CalChans. CalFinalize is doubly templated, taking the CalChan subclasses of the fit input and output as template arguments. Fitting is performed by the 'finalize' function, which takes as input a CalCollection object holding the accumulated data, and a CalCollection object for holding the fit results. It also requires a CalFit object capable of fitting the accumulation CalAddChan subclass, producing the fit result CalChan.

The 'finalize' function loops over the accumulated CalAddChans and runs the CalFit fitChannel function on them. The output channel is the CalChan in the output CalCollection at the same index value as the input collection. For this reason, the input and output CalCollection objects provided to 'finalize' must have the same set of channelIDs. One way of guaranteeing this is to construct the object underlying both CalCollection objects with the same MapAbsDetBrowser.

7.6 CalProcessor

CalProcessor inherits both the interface and implementation of initialization and accumulation from CalAccumulate. In addition, it adds the implementation of CalFinalize by calling down to that class's 'finalize' function. Thus, CalProcessor encapsulates the entire calibration processing functionality in a single class.

CalProcessor is a concrete class doubly templated class, requiring as template arguments the CalAddChan subclass to be used during accumulation and the CalChan subclass that is the output of the fit. The CalProcessor constructor takes as input CalCollection objects for both accumulation and fit results, and a CalFit object capable of transforming the Accumulation CalChan into the fit result CalChan subclass. As with CalAccumulate, the collection objects provided to CalProcessor are owned outside the class.

8 Running Calibration in dataflow

The dataflow system consists of a hardware platform plus a VxWorks-based software environment, which has direct access to the front-end detector electronics. Because of this direct connection and because of the large aggregate compute power present in the dataflow system, it is natural to run calibration in dataflow. Because dataflow is a real-time system, it places additional requirements on the calibration software. All of the classes described in the previous sections have been designed to function in the dataflow environment. In addition, structure is provided by the calibration system to facilitate developing dataflow-based calibration. The classes for this are in the

CalOnline and CalOdf packages, and are described below. The following sections assume familiarity with the dataflow system, which is described in separate documentation.

8.1 Calibration Tagged Containers

Tagged containers are the objects used to transmit data from or to dataflow. The calibration system uses tagged containers for two different purposes, first to store and transmit calibration data, and second to define the configuration of online calibration running. These functions are supported by separate subclasses of odfTC, with corresponding iterators. These classes are kept in the CalOnline package (a dual-use package), because they are used both in the Unix and VxWorks environments. A UML diagram of these classes is shown in Figure 6, and they are described in detail in the following sections.

As with all tagged containers, the calibration containers and the objects they contain must have odfTypeRep type Ids. The type Ids for the calibration tagged containers and all the generic CalChan subclasses in BbrCalib are defined in the **CalOnline** file CalTypeIds.hh. The file CalTypeInit.cc initializes these type reps, making them available from the **CalOnline** library.

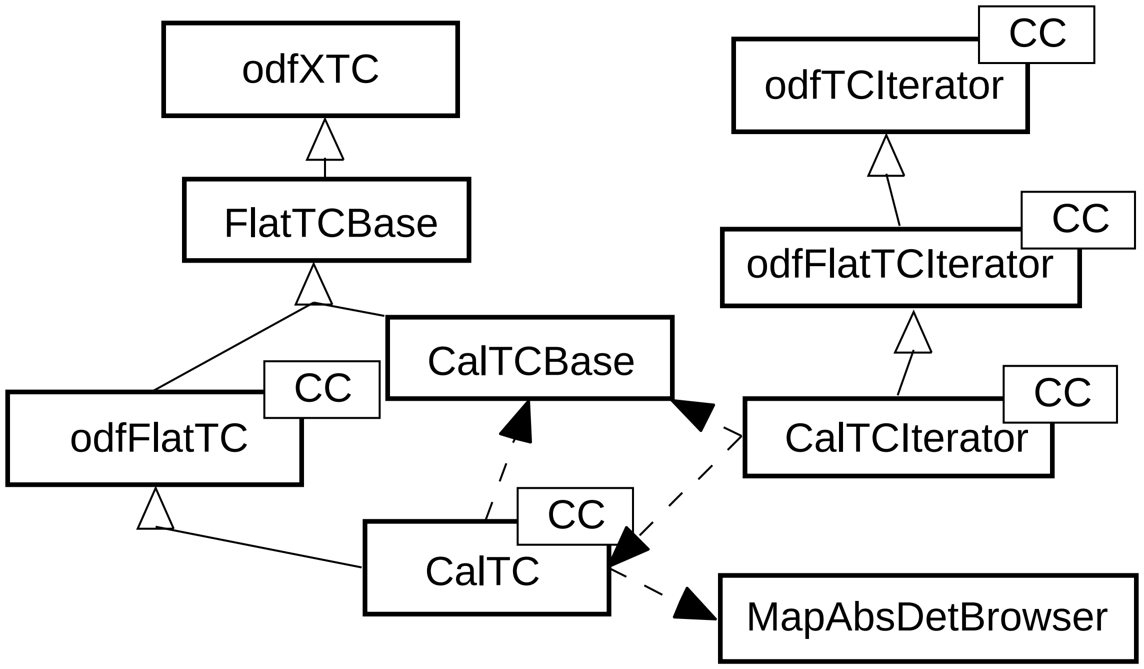

Figure 6 Calibration Tagged Containers

8.1.1 CalTC

CalTC is the tagged container subclass used to transmit calibration data. CalTC defines a collection of CalChans that uses a special kind of memory management in order to facilitate dataflow transmission. CalTC inherits most of its functionality from its base class odfFlatTC, which implements channel storage using a self-relative pointer. CalTC is singly templated on CalChan subclass that it holds. In addition to the channel storage, CalTC owns a CalTCHeader, the online equivalent of CalHeader. Due to coding restrictions on tagged containers, no subclassing of CalTCHeader is supported by the calibration system. CalTC is a fully implemented class, and does not support subclassing.

CalTC has no public constructors. Instead, users create CalTC objects using a static 'create' function that builds the CalTC inside an odfXTC or an odfArena. The create function takes as input a AbsMapDetBrowser and an example CalChan, and builds a collection which has a CalChan object for each node visited by the browser, each set to the corresponding detector address.

CalTC is a special tagged container in that the objects it contains (CalChans) have virtual tables. Virtual table pointers are not transportable, as they are absolute memory addresses. For this reason, when CalTC objects are transported, they cannot be directly used. Transforming a transmitted CalTC into a useable object requires reconstituting the CalChan virtual table pointers on the receiving side. Because of the risks inherent in altering transmitted data, reconstitution is performed by creating a functional copy of the transmitted object.

Reconstitution is performed automatically and transparently by the iterator specialized to work on CalTCs, namely CalTCIterator. CalTCIterator is equipped with sufficient logic to decide when reconstitution is necessary, and will not perform it on CalTC objects created in native memory. CalTCIterator inherits most of its functionality from its base class odfFlatTCIterator. Note that CalTCIterator provides only read access to reconstituted calibration data. Thus calibration data cannot be accumulated after transmission, though they may be fit after transmission.

The class CalTCBase, defined in **CalOnline**, functions as the untemplated base class of CalTC, and is used for CalTC reconstitution. CalTCBase also defines the odfTypeRep of CalTC (the class of the template argument of the CalTC can be determined from its 'contains' field). Because all CalTC objects have the same type ID, several different CalTC objects with different template arguments can be contained inside the same tagged container. This 'mezzanine' feature is useful when it is necessary to transmit several CalTC objects (say accumulated and fit results) at once. CalTCBase has no publicly accessible member functions, and should never be used directly by end users. It is used internally by CalTCIterator during CalTC reconstitution.

8.1.1.1 CalTCCollection

CalTCCollection is a subclass of CalCollection that can wrap a CalTC. This class is used to interface calibrations run in dataflow to the calibration processing classes described in section 7. Like CalTC, CalTCCollection is singly templated on the CalChan subclass being stored. A CalTCCollection can be instantiated on a transmitted CalTC, as internally it uses a CalTCIterator to access the data. Necessarily accumulation is not possible on a CalTCCollection built around a reconstituted CalTC.

CalTCCollection has two constructors; one takes a CalTC object, the other takes only an odfXTC. When using the odfXTC constructor, if the identity and contains of that object do not match CalTC and the CalChan template argument (respectively), the CalTCCollection will not be useable. When using this constructor it is therefore necessary to call the isValid function before using the CalTCCollection object. The odfXTC constructor is useful when building a CalTCCollection on a CalTC found using CalTCFinder, described below.

8.1.1.2 CalTCFinder

CalTCFinder is a subclass of odfTCIterator specialized to iterate over an odfContigXTC containing CalTC objects. . As described in section 8.1.1, CalTC objects are created inside an odfXTC, and an iterator is necessary to find them if that odfXTC is transmitted or passed to another object. As described in section 8.2.2, odfContigXTC is the odfXTC subclass used to store and transmit calibration data in dataflow

CalTCFinder is an untemplated class, and so cannot return templated CalTC objects. Instead, CalTCFinder returns the CalTCs it finds as their base class odfXTC. CalTCFinder verifies (on ‘use’) that the odfXTC pointers returned are truly CalTCs by checking their odfTypeId identity.

In addition to the normal odfTCIterator ‘next’ function, CalTCFinder also provides a ‘find’ function. This function iterates linearly through the CalTCs contained in the odfXTC, and returns the first one whose ‘contains’ field matches the odfTypeId passed to the function. This allows rapid location of a particular CalTC object.

8.1.2 CalCycleTC

An important aspect of online calibration running involves configuring the online for a particular calibration run. The state-machine driver (RunControl or any other calibration-aware odfManager subclass) needs the specific counts of Meta, Macro, Major and Minor cycles to sequence the FSM. In addition to the cycle counts, specific code running in the ROM may require specific data. For example, it may be necessary to increment a front-end DAC every BeginMinor transition. In this case, the code setting the DAC must know how many BeginMinor transitions to expect and what DAC value to set on each. This information must be provided coherently to all parts of the Online when the platform is configured (IE on the configure transition), so that they can work together coherently. For example, the code running in OEP may also need to know what DAC value was set, in order to interpret correctly the output it may receive on the same transition.

In order to provide coherent cycle configuration information to the whole online system, it must come from a unique source. The ultimate source of the cycle configuration information will be the configuration database. However since dataflow does not have direct access to the configuration database, an intermediate form must be used to distribute this data. Our design uses a hierarchy of tagged container objects to describe the cycle configuration data, accessed via a Client-Server protocol. The server will handle the coherent distribution. Tagged containers are convenient for this application since they are transportable to all online systems, have a well-defined and familiar interface, and tagged container objects can be naturally formed into a hierarchy.

The tagged container class used to define and transport cycle configuration information is CalCycleTC. This class inherits from odfXTC, and serves as the base class to define subsystem-specific implementations. An object of class CalCycleTC (or subclass) should contain all the configuration data needed for a particular pair of ‘begin’ and ‘end’ transitions at a particular level of the FSM (Meta, Macro, Major, or Minor) for a particular calibration.

CalCycleTC contains an enum that defines the different levels of the hierarchy. Each CalCycleTC object contains one such enum value, defining its level on construction. CalCycleTC also contains an enum defining different transmission flags. These flags specify what data gets transmitted out of dataflow when the ‘end’ transition of this cycle occurs. More information about data transmission is provided in section 8.2.4.

Subsystem-specific data for configuring a particular cycle should be added to CalCycleTC by subclassing. For instance, to describe configuring a DAC, CalCycleTC should be subclassed to include the DAC setting (and perhaps the DAC register address) as a data member. It could also include configuration information needed for any fitting done on the ‘end’ transition (initial values, parameter limits, fixed parameters, etc.). CalCycleTC subclasses should define their own type Ids, which should be initialized in subsystem specific packages.

CalCycleTC provides tools for building up a cycle hierarchy. CalCycleTC objects can be appended to one another at the level below the object being appended to. Appending several objects below a single object creates multiple steps in that cycle. Checks are made when appending that the cycle level of the objects correspond to their position in the hierarchy. CalCycleTC objects can only be appended if they are contiguous in memory (the entire hierarchy must be contiguous in order to be transmitted): this is also verified on append. An example of how to correctly construct and append CalCycleTC objects is given in the TestCalCycles program in **CalOnline.**

CalCycleTC also provides tools for navigating the cycle hierarchy. The functions next/previous and parent/child allow navigating the hierarchy once it is built. These functions allow generic navigation (i.e. without knowing which CalCycleTC subclass the object really is), a feature which is used to sequence the FSM with a single piece of code for all calibrations. Calibration sequencing is discussed again in section 8.3. CalCycleIterator is a templated odfTCIterator subclass for navigating a single level of the cycle hierarchy. CalCycleIterator is templated on the specific CalCycleTC subclass, and thus returns specific cycle configuration data in a type-safe way. CalCycleIterator should be used instead of the generic navigation tools whenever specific information in a CalCycleTC subclass needs to be accessed.

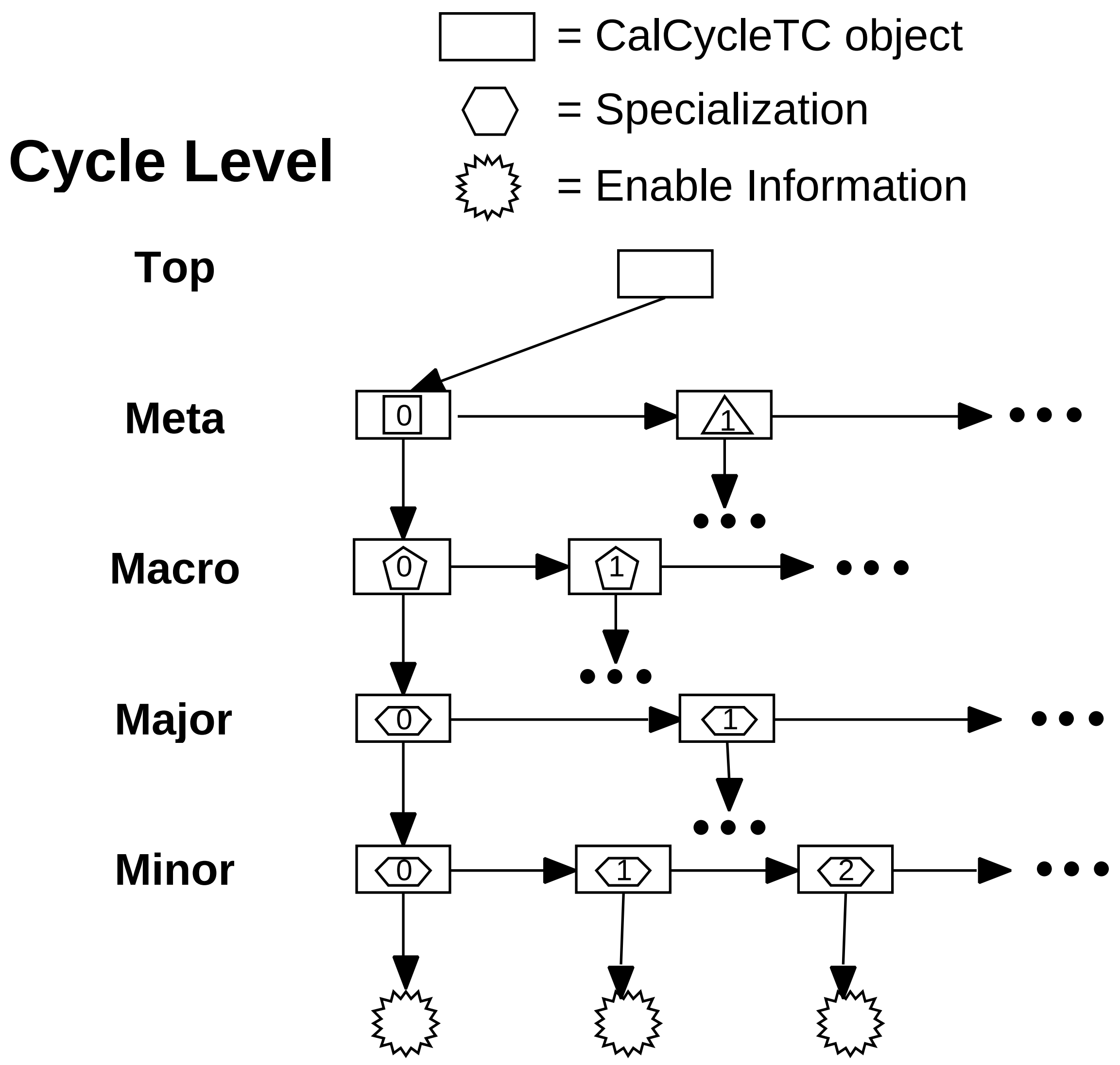

Figure 7 Example cycle TC hierarchy

A diagram of a typical CalCycleTC hierarchy is shown in Figure 7. The top object of the hierarchy is a single CalCycleTC object that contains (via its extent) all the other CalCycleTC objects. This TC does not represent a transition, and is merely used as a handle to the rest of the hierarchy. This top object must explicitly be of class CalCycleTC (not a subclass). Below it are objects for the transitions, arranged in the order the transitions will be executed chronologically. Each CalCycleTC object contains (via extent) the objects below it, making the hierarchy self-similar. This property is used to distribute pieces of the hierarchy during sequencing.

The bottom objects in the hierarchy represent the BeginMinor and EndMinor transitions. Because there is a one-to-one relationship between a minor cycle and the enable transition, this CalCycleTC object must also specify the enable transition parameters. These parameters are defined by the class CalEnableData, which describes the gate source for generating the pulses, and the opcodes and delays for up to three commands to be issued each pulse. The number of pulses generated on ‘Enable’ is specified by the CalCycleTC step count. The minor cycle CalCycleTC object must contain a CalEnableData object in its extent, in order for the FSM sequencing to function correctly. Special minor-cycle creation functions and constructors are provided by CalCycleTC to make this easy. Subsystem-specific subclasses of CalCycleTC can use this same strategy to create minor cycle objects, avoiding the need for a dedicated minor-cycle CalCycleTC subclass.

The program TestCalCycles is built by the CalOnline package. This provides a simple test of the CalCycleTC hierarchy construction and navigation. It creates a correct hierarchy of CalCycleTC objects given user input on the topology, and saves it in binary form (odfSimpleArena) to a disk file. Running the program with the name of an odfSimpleArena disk file containing a CalCycleTC hierarchy as argument causes it to read that file and fully navigate the cycle hierarchy, printing out a message at each node.

More information on the use of CalCycleTC is provided in section 8.2.3 and the following section.

8.2 Calibration Actions

Code intended to run on the ROM for must inherit from the class odfAction. Instances of this class can be attached to specific FSM transitions, with the ‘fire’ method defining the behavior when that transition occurs. To support calibration running in dataflow, an interface layer of calibration-specific odfAction subclasses is provided. These are base classes that must be subclassed to implement specific calibrations. They serve to package the interface between calibration and dataflow into a standard form, and to solve some basic design problems associated with behavior coherence and data distribution. While it is not required to use these calibration action subclasses to implement dataflow calibration, their use is encouraged to avoid duplicating effort spent solving common problems.

To function correctly, calibration requires the action objects attached to different FSM transitions to share data and behave in a coherent way. For instance, the data that are accumulated in a L1Accept transition must be initialized in the preceding BeginMinor transition, and fit in the subsequent EndMinor action. Coordinating the behavior and data between the separate action objects is not completely expressed in the static design of the calibration action classes. Instead, coherence is generated as a dynamic property of how the objects work together when the FSM is sequenced. The relevant classes and their dynamic behavior are described in the following sections. A UML diagram of these classes is shown in Figure 8, and an object diagram relevant for FSM sequencing in Figure 9.

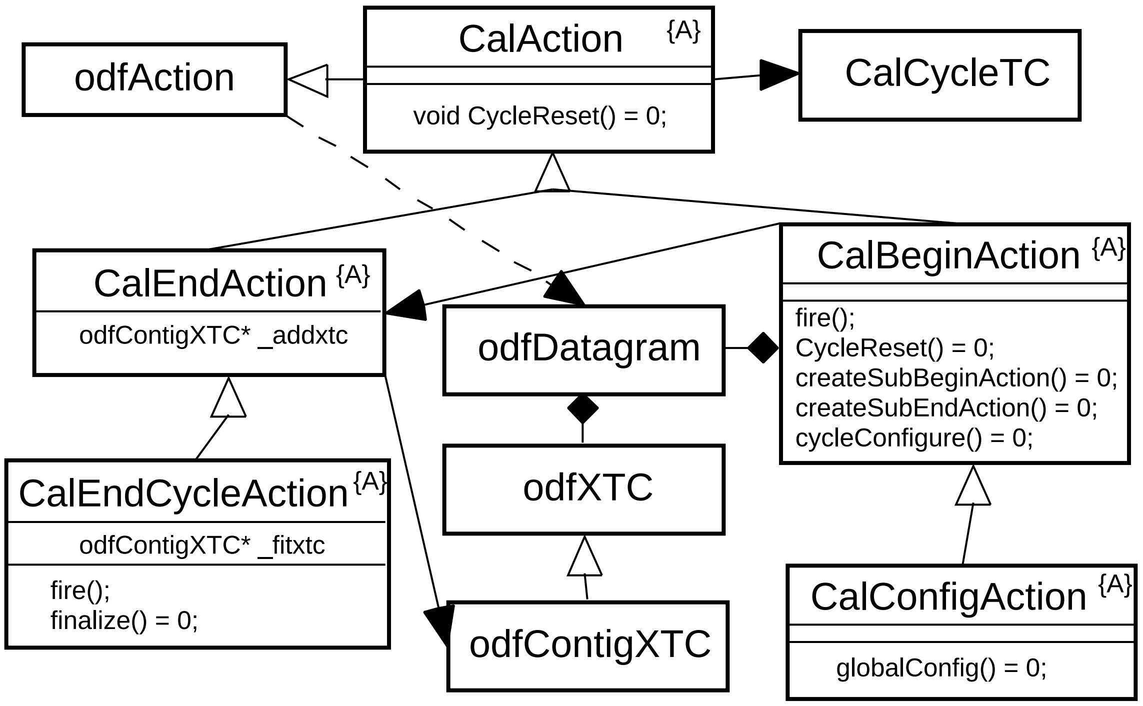

Figure 8 Calibration odfAction subclasses

8.2.1 CalAction

The base class for all calibration actions is the class CalAction. CalAction implements some of the basic odfAction functions, such as defining the duration of time the action requires to perform its work. CalAction also provides access to the relevant configuration data for the particular transition the action is attached to. CalAction defines the cycle level of the action, and provides basic functions for counting the cycle steps.

CalAction contains by const reference a CalCycleTC object. This object is the top of a cycle hierarchy which starts one level above that of the transition to which the CalAction is attached (i.e. a CalAction for minor cycles contains a major-level CalCycleTC). This hierarchy contains all the cycle configuration information necessary to configure all the transitions the action will process. Because the cycle hierarchy is self-similar, the code for accessing the cycle configuration data in CalAction works independent of the transition to which the action was attached. The CalCycleTC object referenced by a CalAction is automatically kept current with the FSM state by base-class functions.

CalBeginAction and CalEndAction are the immediate subclasses of CalAction, used respectively to define the ‘begin’ and ‘end’ transitions. Note that, in the context of calibration, the L1Accept transition is considered to be the ‘end’ transition of the pulse cycle, and the Configure transition is considered the ‘begin’ of the top cycle. This identification allows reuse of code, completely covering the calibration transitions with a minimum number of classes. There is no ‘begin’ transition for the pulse cycle, and there is no ‘end’ for the top.

8.2.2 CalBeginAction

One of the primary jobs of CalBeginAction is to manage the memory used to accumulate and fit calibration data at its level. Memory is managed using a datagram that is a data member of CalBeginAction, created from a standard odfPool. The size of the datagram must be specified on construction, and should be large enough to hold all the CalTC objects that will be accumulated at the CalAction’s cycle level. The odfXTC inside the datagram is actually an odfContigXTC, and is allowed to contain only CalTCs. Nonconst access to this XTC is provided to CalBeginAction subclasses, so that they can create the CalTCs inside it (see section 8.1.1 for details on creating CalTC objects). Note that the datagram owned by CalBeginAction is never transmitted, though the data content of the CalTCs contained in it can be transmitted (see section 8.2.4).

CalBeginAction also controls the behavior and data coherence between the separate actions attached to the FSM. CalBeginAction guarantees coherence by creating itself the ‘begin’ and ‘end’ actions of the cycle level below it, and attaching them to the FSM. This guarantees that the sublevel actions are appropriate for the calibration the CalBeginAction object itself is implementing. By starting with the Configure transition and operating recursively, this implementation insures a fully self-consistent population of the FSM.

CalBeginAction creates the ‘begin’ and ‘end’ sublevel actions using two pure virtual functions, ‘createSubBeginAction and createSubEndAction. These functions should be implemented by subclasses to return a pointer to a CalBeginAction or CalEndAction respectively, created on the heap (i.e. using operator ‘new’). In the event that no action is appropriate for the ‘begin’ or ‘end’ of the sublevel (such as ‘begin’ for the ‘pulse’ level), a null pointer may be returned. CalBeginAction assumes ownership of the objects returned by these functions, and will correctly delete them when it itself is deleted. Through recursion, deleting the Configure transition CalBeginAction object will delete all the CalActions attached to the FSM.

CalBeginAction supports two modes of creating the sublevel actions, specified by an enum value, which must be supplied on construction. The first (the default value) is that the sublevel actions are only created the first time ‘fire’ is called, and reused thereafter. In this mode, subsequent calls to ‘fire’ cause the sublevel actions to be reset to use a new branch of the CalCycleTC hierarchy. The second mode deletes and recreates the sublevel actions each time ‘fire’ is called. This mode is intended to support sophisticated calibration runs that dynamically reprogram the FSM.

CalBeginAction also supports the notion of skipping levels in the cycle hierarchy. Given the design of the FSM, the depth of the cycle hierarchy is fixed. The number of cycle levels was chosen to be deep enough even for elaborate calibrations, making it too deep for most simple calibrations. Without skipping, the recursive CalBeginAction action creation scheme would force every calibration implementation to provide classes for every level of the hierarchy, even if there was nothing specific the calibration needed to do at that level.

Level skipping is allowed through specification of the dLevel parameter on construction. The default value of dLevel is 1, signifying that the CalActions returned by createSubBeginAction and createSubEndAction should be attached 1 level below the current action. Specifying a larger value of dLevel causes the actions be attached further down in the FSM. Note that the level of the returned CalActions must agree with their intended attachment level. The default odfAction originally attached to the FSM on its construction is left in place for skipped levels. Transitions for skipped levels still occur, but no time is spent on them since they have no real code attached.

CalBeginAction distributes the relevant portion of the CalCycleTC hierarchy down to the create* functions, allowing subclasses to use the cycle configuration data when constructing those objects. The sublevel ‘begin’ action is created first, then the ‘end’ action. CreateSubEndAction has two arguments not present in createSubBeginAction, namely the XTC of the CalBeginAction calling the function, and the XTC of the sublevel CalBeginAction just created. Section 8.2.4 describes how these extra arguments are intended to be used.

CalBeginAction fully implements the odfAction ‘fire’ method. When ‘fire’ is called, CalBeginAction checks to see if it needs to create ‘begin’ and ‘end’ actions. If so (on the first call to ‘fire’, or if the subaction fate is to be recreated), it creates them using the functions described above, and attaches them to the FSM at the specified sublevel to itself. If the actions don’t need creating, it simply resets the existing actions. CalBeginAction ‘fire’ also calls the pure-virtual function ‘cycleConfigure’. This function should be implemented to configure the hardware and software specific to the calibration being run. Subclasses can access the configuration data in the CalCycleTC hierarchy to do this.

8.2.3 CalConfigAction

CalConfigAction is a subclass of CalBeginAction, and defines objects intended to be attached to the Configure transition. CalConfigAction functions as a CalBeginAction at the ‘top’ level, creating the ‘begin’ and ‘end’ actions nominally at the Meta level. In addition, CalConfigAction processes the Configure transition, retrieving both the general and the cycle-dependent configuration data specified the environment key, and performing the global (non-cycle-dependent) configuration of the hardware. As with all the calibration actions, CalConfigAction must be subclassed for a particular calibration implementation.

CalConfigAction implements the ‘cycleConfigure’ function defined by CalBeginAction, using this to retrieve the cycle configuration data through the ‘fetchCycleTCs’ function. This function is currently implemented to read an odfSimpleArena file defined by the configuration key. The exact file name required by CalConfigAction is of the form [Xxx]Cycle[FFFFFFFF].dat, where Xxx is the three-letter acronym of the subsystem, and FFFFFFFF is the hex representation of the configuration key value. Note that an (optional) arbitrary directory prefix for where to find this file may be specified when constructing a CalConfigAction object. CalConfigAction assumes ownership of the CalCycleTC hierarchy returned by fetchCycleTCs. The ‘cycleConfigure’ function then calls ‘globalConfig’, a pure-virtual function intended to configure those parts of the hardware and software which don’t depend on the cycle level or step.

CalConfigAction also defines what happens to L1Accept data after leaving dataflow. By default, CalConfigAction will sink the L1Accept data, not transmitting it outside dataflow, thereby reducing the ROM overhead. This default can be overridden by calling the ‘forwardL1Accept’ function in a subclass globalConfig implementation. Calling this function will result in L1Accept data being sent to the event level (normally OEP). The inverse function ‘sinkL1Accept’ is also provided. These are both protected.

8.2.4 CalEndAction

CalEndAction specializes CalAction for the accumulation of calibration data. The CalEndAction constructor takes a reference to an odfXTC, passed to it by the createSubEndAction function of the CalBeginAction object above it in the cycle hierarchy. This odfXTC is owned by the CalBeginAction (see section 8.2.2), and contains the CalTCs which the CalEndAction is intended to accumulate. To find the CalTCs inside the odfXTC, CalEndAction subclasses can use the CalTCFinder described in section 8.1.1.2, or they can be passed explicitly to the subclass constructor.

Unlike CalBeginAction, CalEndAction doesn't implement the 'fire' method. This allows subclasses intended for the L1Accept transition .to implement 'fire' without any unnecessary overhead. CalEndAction subclasses intended for 'end' transitions other than L1Accept should inherit from CalEndCycleAction, as described in the next section.

8.2.5 CalEndCycleAction

CalEndCycleAction further specializes CalEndAction for implementing non-L1Accept 'end' transitions. Like L1Accept actions, 'end' actions are responsible for accumulating data into the cycle level above them (i.e. the endMinor action accumulates data for the Major cycle). Additionally, non-L1Accept 'end' actions must fit the data already accumulated in their own cycle level. CalEndCycleAction provides the framework for this additional step.

The CalEndCycleAction constructor takes as input the odfXTC owned by the 'begin' action of its own level, in addition to the odfXTC owned by its parent 'begin' action. The odfXTC from its own level contains the CalTCs it is intended to fit, while the other contains the CalTCs it is intended to accumulate. These odfXTCs are passed to the constructor through the createSubEndAction function of its parent level CalBeginAction. The CalTCs that the CalEndCycleAction is intended to fit or accumulate can be found in the respective odfXTC objects using the CalTCFinder described in section 8.1.1.2.

Like CalBeginAction, CalEndCycleAction fully implements the 'fire' transition. This handles all the cycle count bookkeeping and the memory management associated with data transmission, and calls the pure virtual function 'finalize'. The 'finalize' function is intended to implement both the fit of the cycle's own data as well as the accumulation of the cycle's parent's data. Fit results should be put in the odfXTC that is passed to the 'finalize' function. The fit status should be returned as the return value of the function.

The odfXTC passed to 'finalize' is contained in a datagram, owned by CalEndCycleAction. If the action isn't transmitting any data, this datagram is reused every transition. If the action does transmit data, the datagram is remade each transition, as ownership of datagrams is passed to dataflow on transmission. Because transmission is determined by configuration, the odfXTC passed to 'finalize' may or may not contain the CalTCs the action is intended to fit. Therefore, subclass implementations of 'finalize' should always check if their fit result CalTCs are already present in the odfXTC when it is passed to them, and only create the CalTCs if they cannot be found.

CalEndCycleAction fully implements data transmission. Depending on the level of diagnostics required, on a given 'end' transition the CalEndCycleAction may be required to transmit accumulated data, fit data, both, or none.

CalEndCycleAction uses the value of the transmission flag of the current CalCycleTC to tell it what data to transmit on a given transition (see section 8.1.2 for more details on transmission flags). If accumulated data are to be transmitted, the contents of the accumulate XTC are copied into the output datagram before the call to 'finalize'. The cumulative size of both the accumulation and fit CalTCs (with some buffer) must be provided to the CalEndCycleAction constructor. This insures that the datagram returned on 'fire' will be large enough to handle any value of the transmission flag.

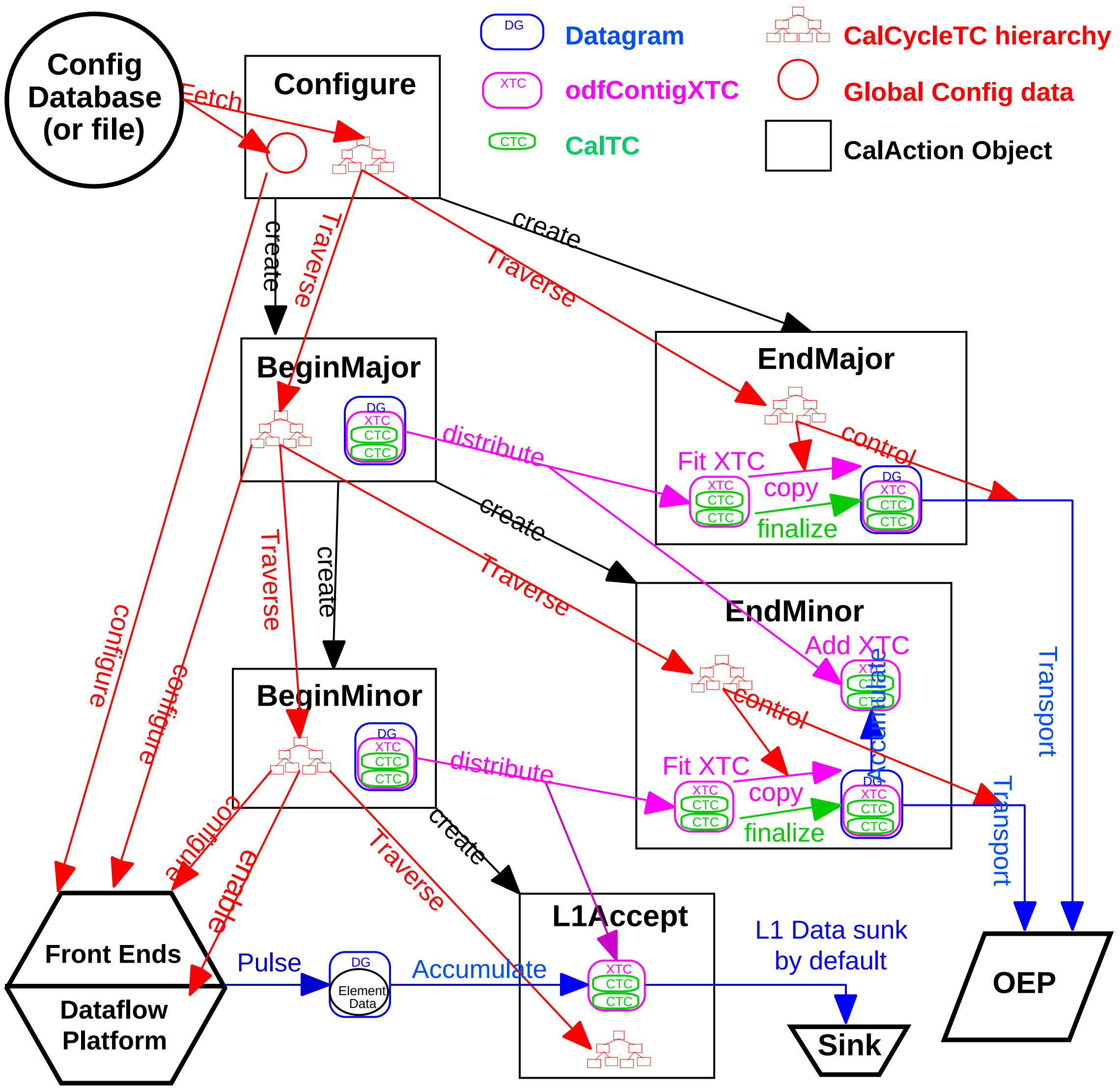

Figure 9 CalAction Objects for a major-level calibration

8.3 Calibration FSM sequencing

To sequence the online FSM according to the transitions implied by a CalCycleTC hierarchy, a specialized subclass of odfManager is needed. Eventually the job of traversing the cycle hierarchy will be performed by the dataflow proxy running inside RunControl. In the mean time, simple command-line programs for this task are provided by building the control level of the **CalOdf** package.

The programs created in **CalOdf** are called TestCalManager and RunCalManager. The 'Test' program sequences a standalone UNIX FSM, invoking only dummy actions. The 'Run' program sequences the real online FSM, and requires running source, segment, and event level processes to function correctly. Both programs present the same user interface, fully sequencing the FSM from 'Map' to 'Unmap' once supplied with the hex value of the configuration key corresponding to a valid CalCycleTC hierarchy file (see section 8.2.3). Both programs can accept a valid hierarchy of arbitrary CalCycleTC subclasses, provided the top level object is literally of class CalCycleTC, and the minor-cycle objects all contain a CalEnableData object.

The prefix of the CalCycleTC hierarchy file may be specified on the command line of either program. The default prefix is 'Orc'. To use the same file as that read by CalConfigAction, subsystem users may wish to provide '[Xxx]Cycle' as prefix. An arbitrary directory path may also be prepended to this prefix.

CalCycleManager is the odfManager subclass equipped to sequence the FSM according to a CalCycleTC hierarchy, and is used internally by both TestCalManager and RunCalManager. This class implements the file parsing and hierarchy traversal. End users never need to deal directly with this class.

8.4 Calibration template instantiation in dataflow

A general problem encountered when using calibration inside dataflow has to do with template instantiation. Because the VxWorks C++ compiler does not correctly support automatic template instantiation, it is necessary for users to instantiate the calibration templates they use in their own code. Templated classes are used extensively in the calibration system, and many user-level classes invoke sub-templates, creating a large chain of templates that must be instantiated. In order to ease the burden on subsystem developers and shield them from implementation details, a set of macros have been defined which package the template instantiation into a clean form. These macros are kept in the **CalOnline** package in the file CalTemplateMacros.hh. Note that the **CalOnline** package itself instantiates a few templates (in CalTemplates.cc) which are needed for the functioning of that package, but users are required to instantiate their own templates, even when using generic CalChan and CalCycleTC subclasses. The issue of dynamically loading calibration libraries with templates will be addressed in a subsequent calibration release.

The following macros are defined in CalTemplateMacros.hh:

- ♦ CalChanMacro: instantiates all templates needed for normal dataflow operation given a CalChan subclass (instantiates CalTC, CalTCCollection, and CalTCIterator)
- ♦ CalAddChanMacro: Instantiates all templates needed for normal dataflow accumulation given a CalAddChan subclass and the argument it uses for accumulation (Instantiates CalChanMacro and CalAccumulate)
- ♦ CalHistChanMacro: instantiates CalAddChanMacro plus histogram-specific templates.
- ♦ CalProcessorMacro: Instantiates all templates needed for normal dataflow operation given the accumulation and fit CalChan subclasses, plus the accumulation argument of the CalAddChan (instantiates CalAddChanMacro, CalChanMacro and CalFinalize)
- ♦ CalCycleMacro: Instantiates all templates needed for normal dataflow operation given a CalCycleTC subclass (instantiates CalCycleIterator)

These template macros should not be instantiated inside of other classes. Instead, they should be put in global functions or in standalone .cc files compiled into object libraries against which the classes that need them are linked.

9 Persistent Calibration Classes

CalDatabase holds the base classes for the calibration persistent objects, and the transient objects with which end users interact with them. The transient proxy also defines the validation, verification, and storage user interface. These classes are described in the following sections.

The base classes used to store calibration data, and to describe the conditions under which the calibration data were accumulated, are defined in the **CalDatabase** package. A UML diagram of these classes is shown in Figure 10, and they are described in detail in the following sections.

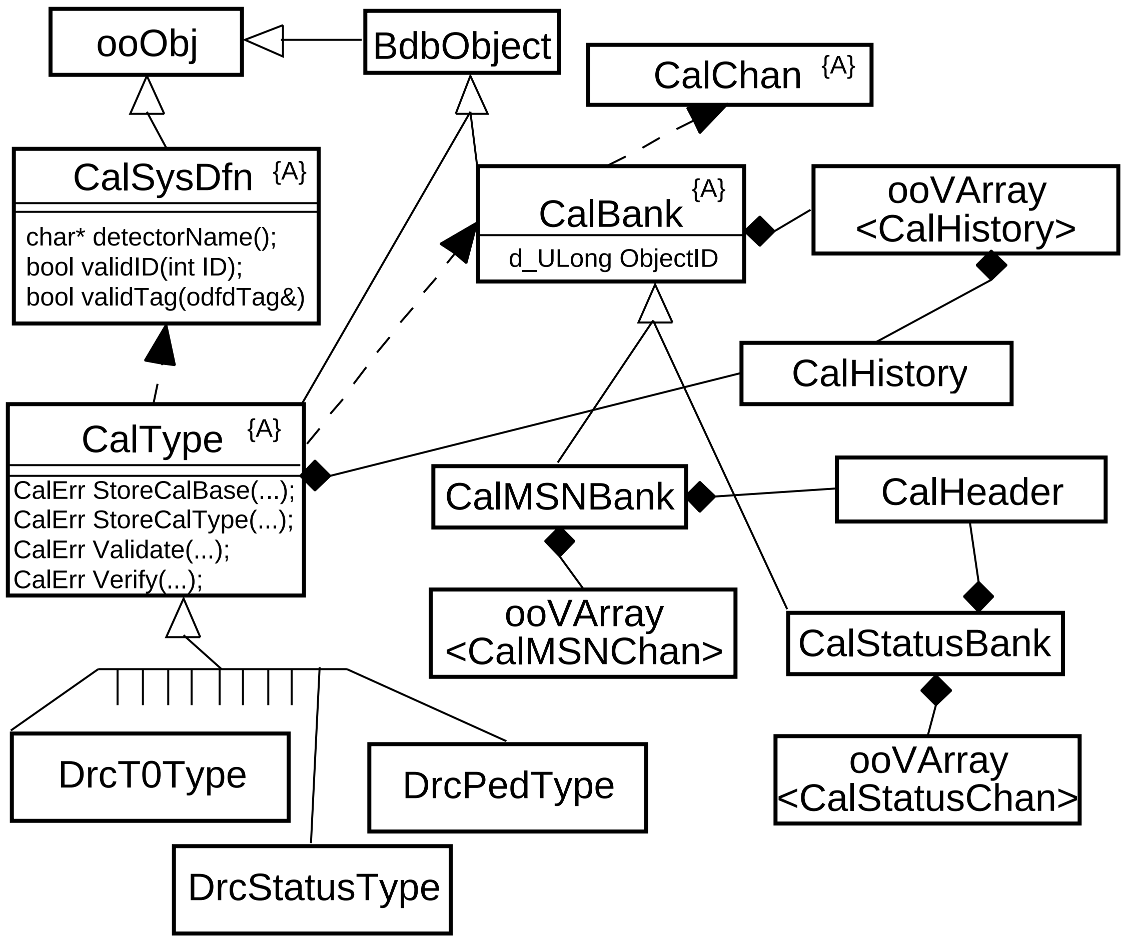

Figure 10 Persistent Calibration classes

9.1 CalBank

This abstract base class defines the CalChan collection class that can be stored in the database. CalBank is a persistent-capable class, so all its subclasses can be persistently stored. Storing channels as collections is necessary since Objectivity overhead (16 bytes/object plus access time) precludes storing channels as individual objects. Since most use cases involve manipulating channel collections, this should not be a problem.

CalBank and its subclasses are not intended to be used directly by end users, as this could put persistent data in jeopardy. Consequently, the interface functions to this class are extremely sparse to discourage its use. End users wishing to access calibration data should use one of the CalBase subclasses defined below, or a dedicated offline proxy.

CalBank subclasses are associated with specific CalChan and CalHeader subclasses, but not with specific calibration types. Consequently, the same CalBank subclass can be used to store calibration constants for several calibration types in different subsystems. As with CalChan, generic CalBank subclasses should be used whenever possible to reduce the proliferation of classes.

CalBank subclasses own their CalChan data through the persistent variable-length array class ooVArray. Because the ooVArray template does not support polymorphism, it cannot be declared in the CalBank base class. Instead, CalBank subclasses must each declare an ooVArray for the specific CalChan subclass used by the type[1]. This is demonstrated in the generic CalBank classes described in section 4.2. The CalBank base class contains an ooVArray of CalHistory objects to record the history of how the object was generated.

In general, several CalBank objects are needed to define the constants for all channels in a subsystem (IE the channels are divided among several CalBank objects). The division of channels into banks is defined by the CalType class, which defines the calibration type, as described below. Assignment of channels to banks cannot be arbitrary, as the ability to download calibration data back into dataflow currently requires that channels belonging to a ROM not be split between objects. CalBank channel grouping is discussed in more detail in section 9.3.

9.2 CalType

CalType is an abstract base class whose subclasses define calibration types. This class does not itself contain calibration data; rather it defines the conditions that calibration data must satisfy in order to be stored. As such, a single CalType object can be associated with an arbitrary number of actual calibration data objects.

The CalType class performs a number of functions, namely:
♦ Gives the calibration type a unique name (its class name)
♦ Defines the CalChan and CalHeader subclass used for this calibration type
♦ Provides a physical interpretation of the CalChan fields
♦ Defines the significance of the public CalChan and CalHeader flag fields (if these are used)
♦ Defines the algorithms and cut values whereby new calibration data are verified and validated
♦ Stores new calibration data
♦ Defines validation conditions for storing new CalType objects

[1] Objectivity version 5 will allow classes to define ooVArrays using a template argument, removing the requirement that CalBank be subclassed for every CalChan subclass.

♦ Stores new CalType objects

Much of this functionality is fully implemented in the base class, so calibration developers need only implement a few virtual functions. End users need never be aware of the existence of CalType objects, as they function automatically behind the calibration public interface. A detailed description of the CalType functions is given in section 15.4.

Unlike CalChan and CalBank, there are no generic CalType classes: each calibration type must have its own unique CalType subclass. Calibration types which describe essentially the same information obtained via different mechanisms (EG ECAL gain measured via source or pulser) can and should share the same CalChan, CalHeader, and CalBank classes, but must have different CalType subclasses (though they could inherit from a common base class). Unlike CalBank, a single CalType object describes all channels in a subsystem.

CalType is a persistent capable class. The base class data members describe the basic parameters of the type and the structure of the data storage for the type (which CalBank class, CalChan class, the number of objects for this type, etc.). Subclass data members are intended to store the values for cuts used in the channel content and CalType content tests. CalType are expected to be updated much less frequently than CalBank objects, as they describe detector configuration, not detector data. CalType owns a CalHistory object, in order to record its creation.

One of the jobs of a CalType object is to allow safe storage of new calibration data presented in the form of a CalBase object. Consequently, a CalType object must be present in the database before data of its calibration type can be stored. Before copying the CalBase content into a persistent CalBank object, the CalType validates both its structure and content (validation can also be performed as a separate step before storing data). The structure test insures that the data are consistent with the calibration type (correct CalChan and CalHeader subclass), and that the channel IDs correspond exactly to those of the specified CalBank object. This test is implemented in the base class, guaranteeing that all stored calibration data are structurally correct.

Another CalType job is to verify data. Verification differs from validation in that it tests whether new data are consistent with old, whereas validation makes the stronger test of whether new data are fit to replace old. Verification will be used to make factory-mode running of BaBar more efficient, by avoiding the costly overhead of database storage and dataflow download of new calibration constants when the old ones are still adequate. The CalType verification interface is very similar to the validation interface, having a mandatory structure test with a user-defined content test. The verification structure test is identical to the validation test. The content check for verification is however separate from validation, and may or may not use the same algorithms and/or cut values.

The responsibility for defining both validation and verification content checks for every calibration type lies with the subsystem developer. A fully implemented template function for the CalType base class provides the content checks. This template compares each channel with the most recent data for that channel taken from the reference calibration of this type. The general channel status flag for this channel can also be considered when making this comparison, to allow known bad channels to be skipped. The CalBase is considered to have been validated/verified if the number of channels failing this comparison test is less than a value defined per-object by the CalType object (this threshold is a base class data member). To account for drastic changes in calibration constants that may occur for instance when hardware is exchanged, the content test may be overridden via an optional argument.

During the validation content check, the CalType object is allowed to change the channel data. This might be necessary for instance if the data are sufficiently distorted that they pose a danger to dependent downstream code. In this case, a reasonable strategy would be to replace distorted data with valid data for the offending channel from the reference set. The CalChan flags should always be set according to the original data. The verification content check is not allowed to change the channel data.

CalType do not own the data used to check the CalBase structure. Instead, they reference it through a CalSysDfn object. All CalType of a given type share the same CalSysDfn, though different types may refer to different CalSysDfn objects. The association of a CalSysDfn object with the first stored object of a new CalType must be made after constructing that object. Subsequent objects of that type will pick up the correct CalSysDfn reference automatically on construction.

CalType are also responsible for storing new CalType objects for their own calibration type. Before being stored, the content of each new CalType object must be validated by the most recent CalType object in the database. The responsibility for defining the CalType validation algorithm lies with the subsystem developers. The first CalType object stored for a particular type is not validated.

Because CalType objects record the conditions under which calibration constants were stored, it is not allowed to store a CalType object for a time in the past. Every new CalType object is stored with the current program time, defining the tests and conditions pertinent to validating and verifying subsequent calibration data of its type.

9.3 CalSysDfn

CalSysDfn is an abstract base class that defines the grouping of channels into CalBank persistent objects. CalSysDfn is a persistent capable class that is stored outside the normal time interval structure of the Conditions Database, as, unlike other conditions information, it is not allowed to evolve with time. If the dataflow structure of a system changes it will be necessary to create a new database to store new data. No code will necessarily need modification to make this change. The need to have different databases for different system configurations is already foreseen in the general structure of the conditions database.

CalSysDfn must be subclassed for each subsystem. Thus every **XxxCond** package must define at least one XxxSysDfn class. CalSysDfn is flexible enough that the same class can be used to define test beams, test stands, and the IR2 configurations. Different configurations will require different objects of this class.

The information content of a CalSysDfn object is somewhat redundant with that of the Odf persistent map. To avoid storing redundant persistent information, it may be desirable for subsystems to implement their CalSysDfn object using persistent map objects. This may not be the most efficient implementation of the CalSysDfn functions, and so is not required.

CalSysDfn uses the dataflow detector tag class odfdTag to validate channel structure. To be valid, the channel IDs of the CalChans must correspond to a valid detector tag for that subsystem.

9.4 Persistent Object Virtual Table Loading

Objectivity supports polymorphic references and handles to persistent data, a feature used extensively in the calibration system. While very useful, this feature can cause puzzling run-time errors if special care is not taken at link time. The problem comes because, in retrieving objects, Objectivity does not call constructors. Thus, if a retrieved persistent object is referenced polymorphically, the loader may see no reference to the actual subclass of the object, and therefore omit the subclasses specific functions and virtual table from the executable. The symptom for this is a segment violation during the call to a virtual function of a persistent object.

To solve this problem the programmer must insure that, somewhere in their executable, an object of every persistent class used in that program is created by explicitly calling the concrete classes constructor. Additionally, Objectivity recommends that a virtual function of that class be called for that object. This object should not be created persistently or on the heap, and the program execution flow can even skip the virtual function call, so long as the loader has no way of knowing the call will be skipped

To facilitate and standardize the practice of virtual table loading, the class CalVTblInit is provided in the **CalDatabase** package. This class defines a virtual interface for initializing persistent class virtual tables. CalVTblInit was developed to implement the calibration database browser, but it can be used in any context where virtual table loading is an issue.

10 Calibration Proxies

Because Objectivity intrinsically provides no protection against persistent data deletion, modification, or addition, all user interactions with calibration persistent data is buffered by transient proxy classes. These classes contain the same information content as the persistent objects they proxy, but can be safely used in user code without the risk of destroying or modifying archival data. In addition, these transient classes perform data quality and consistency checking necessary to guarantee the quality of newly entered calibration data. A UML diagram of these classes is shown in Figure 11, and they are described in the sections below.

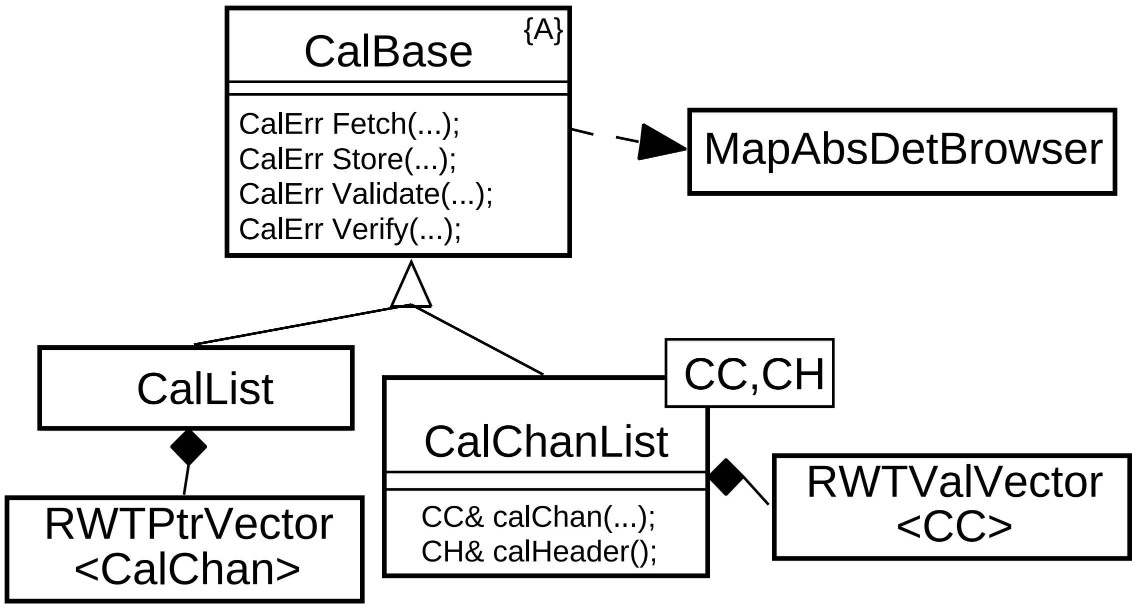

Figure 11 Transient Proxies for Persistent Classes

10.1 CalBase

CalBase is an abstract base class that defines the calibration user interface for verification, validation, and data manipulation. It also implements *store* and *retrieve* functions, which allow users to move data in and out of the calibration database. A list of CalBase functions can be found in section 15.3.

A CalBase object is a transient proxy to a CalBank. Data stored in the conditions database in the form of a CalBank can be loaded into a CalBase, where it can be manipulated and viewed. Conversely, data in a CalBase object can be stored in the database as a CalBank object, subject to passing the validation checks. This avoids users ever having to code directly to CalBank, which could compromise persistent data. CalBase objects can be used anywhere in user code where generic access to calibration data suffices. Offline applications that require read access to specific calibration data should probably implement a dedicated proxy to their CalBank data instead of using a CalBase subclass.

CalBase also defines the online calibration interface between dataflow and OEP. Data generated in dataflow in the form of a CalTC must be converted into a CalBase object in order to be manipulated and/or stored in the database during OEP processing (see section 11.1). The dataflow calibration classes are described in section 8.1.1.

Each CalBase object contains a collection of CalHistory objects that summarizes the modifications that have been performed on its data. All CalBase operations that alter calibration data automatically add a CalHistory object that describes the action, creating a log of how the data was created. Additional CalHistory objects can be entered by the user as desired. Because the CalBase collection of CalHistory objects is stored when the CalBase is stored, a complete record of how calibration data was generated or modified will always be available.

CalDatabase contains three concrete implementations of CalBase, two of which are described below. The third, CalIndirectList, is described in section 12.4. A subclass of CalBase intended for use in the multi-processor Prompt Reconstruction environment is foreseen. Details on this class will be given in a later note.

All the concrete CalBase subclasses have constructors that take a CalMapBrowser object as input, and use it to create a collection whose CalChan members have the channelID numbers specified by the browser. These constructors also take as argument an 'example' CalChan object, whose Chan subclass specific state (IE everything outside the channelID and the status flag) is copied into every channel in the collection.

10.2 CalList

CalList is a CalBase subclass, which uses polymorphic storage of the CalChan and CalHeader objects it, contains. CalList uses the Rogue Wave pointer sorted vector template to store an ordered (by channelID) list of CalChan pointers. This class adds no new functions on top of the CalBase interface. Its use is appropriate where generic access to channel data is sufficient. For instance, CalList is used inside the CalType base class to access status information.

10.3 CalChanList

CalChanList is a templated CalBase subclass that provides a type-safe interface to channel and header data in addition to the polymorphic CalBase interface. This class is appropriate for use when the type of calibration data being used is known at compile time. CalChanList is also used as the underlying storage in the CalListCollection described in section 7.1.

11 Offline Tools

A number of tools have been developed to facilitate calibration in the Unix environment. These are kept in the **CalOffline** package, and are described below.

11.1 CalTC Conversion

A common use case is to convert a CalTC transmitted from dataflow into a CalChanList in OEP. This is necessary to store the results of a dataflow-based calibration in the database. The factory class CalTCConverter, found in the **CalOffline** packge CalTC, can perform this conversion. This class is templated on the CalChan subclass contained in the CalTC. The 'convertTC' function takes as input a CalTC (passed by pointer as an odfTC*), and a CalBase object. The CalBase object must be compatible with storing CalChans of the type specified by the template argument. The 'convertTC' function uses a CalTCIterator to access the channel data in the CalTC, allowing conversion of transmitted data. The iterator is stepped through all the channels in the CalTC, copying them into the CalBase. Channels already stored in the CalBase are not removed. The 'convertTC' function returns a CalErr object, allowing it report errors (for instance, if the odfTC identity and contains do not match a CalTC holding the template argument CalChan subclass).

11.2 Histogramming Tools

It is frequently necessary to examine calibration data graphically and manipulate it interactively. Instead of creating specialized graphics and manipulation tools, the calibration system provides tools to convert calibration data into Heptuple format, allowing the converted data to be viewed and manipulated with standard interactive tools like Paw and MinFit.

Two manager classes, CalBookHistManager and CalBookTupleManager, provide an interface between CalBase objects and lists of histograms or ntuples. These use other classes described below to convert CalBase collections of CalChans into Heptuple histograms and ntuples. CalBookHistManager and CalBookTupleManager contain a reference to the HepTupleManager object they were constructed with and a list of pointers to the ntuples/histograms that they created.

Individual CalChan objects are converted into Heptuple objects by a set of wrapping classes. The class CalHistChanBook is used to store a single object of a CalHistChan subclass into a histogram. The class CalTupleChanBook uses the CalChan data fields to create an ntuple having one column per field, one row per channel. In addition to this generic behavior, these classes can be subclassed to provide specialized behavior when converting specific CalChan subclasses.

The classes CalHistListAccum, CalHistListBook, CalTupleListAccum and CalTupleListBook create collections of Heptuple objects from a CalBase collection of CalChan objects. These classes provide a wide range of options for sorting histograms and ntuples into HepTupleManager subdirectories. For example, in many cases the channel ID is the value of the detector address (odfdAdr) corresponding to that channel. In this case it's possible to use one of the constructors of CalHistListAccum to create a directory tree containing module, section, chip and channel levels. Users can modify the tree if necessary.

Examples of how to use this package are given in the TestCalBook.cc test program that is included in the **CalOffline** package.

12 Indirect Calibration Types

In some calibrations, the underlying electronics may force the results to take on a limited range of discrete values. A hypothetical example of this would be a calibration whose result represents the setting of a small-range (say 4-bit) DAC. Similarly, a-priori conditions (such as geometry) may cause some calibrations to produce results that are equivalent across large collections of channels. In both these cases, the groupings of channels which give equal or equivalent results probably do not fall naturally into the dataflow hierarchy (IE, not all channels in a chip have the same values). In several known cases in BaBar, storing this ‘redundant’ data for every channel results in a significant, unnecessary burden on the conditions database.

The CalIndirect classes were designed to avoid storing redundant channel data. The idea is to provide a layer of indirection, which allows an arbitrary subset of channels within a system to ‘point’ to the same CalChan data object. This indirection is supported by special subclasses of CalChan, CalHeader, CalType, and CalBase, as described below A UML diagram of these classes is shown in figure ***.

12.1 CalIndirectChan

CalIndirectChan is a fully implemented subclass of CalChan that defines the 'indirection layer' of an indirect calibration type. CalIndirectChan has a single integer specific data member (called data index) whose value specifies the channelID of another CalChan (of unspecified subclass). The value of the data index has no relation with the channelID of the CalIndirectChan. CalIndirectChan is defined in the BbrCalib package.

12.2 CalIndirectHeader

CalIndirectHeader is a subclass of CalHeader specialized for use with a collection of CalIndirectChans. CalIndirectHeader has a specific data member that records the maximum index value of the collection to which it is associated.

12.3 CalIndirectType

CalIndirectType is the CalType subclass associated with indirect calibration types. CalIndirectType is an abstract class, and must be subclassed for every unique indirect calibration type. To correctly store and use an indirect calibration, its type must inherit from CalIndirectType. CalIndirectType inherits most of its functionality from CalType. It adds to CalType data members that record the CalChan and CalHeader subclass names of the 'data layer' of the indirection. The CalIndirectType class is defined in the CalDatabase package.

A single CalIndirectType class manages the verification, validation, and storage of both the indirection and the data layers of a CalIndirectList. As in the CalIndirectList 'store' and 'fetch' interface, these layers are distinguished by the value of the objectID argument, with normal values indicating the indirection layer and a special value (-1) indicating the data layer.

To handle their overloaded meaning, CalIndirectType overwrites the CalType 'store','validate', and 'verify' functions. When the objectID value indicates that the input CalBase represents the indirection layer, these functions simply call down to the CalType base functions. When the objectID indicates that the input CalBase represents the data layer, the normal structure check that the channelID values correspond to valid detector tags is bypassed, and instead the channelIDs are required to equal their index value in the CalBase collection. In both cases content check functions are also invoked.

The content check for the indirection layer of a CalIndirectType is fully specified in the base class. This simply confirms that the index values specified in the CalIndirectChans correspond to valid values in the associated data layer. Implicitly, this requires that the data layer be stored before any of the indirect layer objects. The content check for the data layer is the responsibility of the specific subclass implementation. This content check should verify that the dimension of the data layer is sufficient to satisfy all the CalIndirectChans in all objects of the indirection layer, in addition to an appropriate test of the data value contents.

12.4 CalIndirectList

CalIndirectList is a subclass of CalChanList specialized for dealing with indirect calibration types. CalIndirectList is templated on the subclass of CalChan that stores the real calibration data. In addition, CalIndirectList contains a distinct CalChanList fully templated on CalIndirectChan and CalIndirectHeader. CalIndirectList satisfies the CalBase interface using its CalChanList base class. The channelIDs of the CalIndirectChans have valid detector tag values, as with normal calibration types. The channelIDs of the templated CalChanList CalChans must equal their index position in that list, and thus do not correspond to valid detector tags. The value of the CalIndirectChan's data index corresponds to a valid index in this 'data layer' list. Thus a CalIndirectList connects a detector tag channelID with the data contents of the 'data layer' CalChan. CalIndirectList also provides access to the 'indirection layer' CalIndirectHeader. The CalIndirectList class is defined in the CalDatabase package.

CalIndirectList inherits its CalBase interface implementation from CalChanList, which provides direct access to the 'data layer' via their index (= channelID). In addition, CalIndirectList provides two other interfaces to channel data. The first provides access to the CalIndirectChans 'indirection layer' via their index and channelID. The second navigates the indirection to return the value of the 'data layer' CalChan given the index or channelID of the CalIndirectChan. This latter interface allows one to hide the indirection from user code.

13 Calibration Configuration

14 Calibration Database Browser

15 Appendices

15.1 CalChan flag fields

The CalChan flag is divided into private and public fields. The private fields can only assume predefined values, as they are intended to have a universal meaning across calibration types and subsystems. The public fields can be defined for each calibration type, and are intended to describe the results of the verification tests. Flag fields should be defined so that a null value means the channel is 'normal'.

The value of the public flag may be accessed as an integer via the CalChan publicFlag method. It may be set using the setPublicFlag (sets value to input) or addPublicFlag (performs binary OR of the existing field with the input).

The private flag fields are defined in the following list. They can only be set by using the CalChan functions. In general, this should be done only during validation. The values listed below may not be changed, but new values can be added. The enums and bit fields for these are defined in **BbrCalib**/CalFlags.hh.

- Channel Validity. This describes whether the channel has passed the validation test, and can assume the values *Validated* or *NotValidated.* It's the only field whose CalChan construction default value (*NotValidated*) is not 'normal' for data retrieved from the database. This field can only be set by the CalType base class.
- Channel Status. This can assume the values *Working, Disconnected,* or *Suppressed.* In general, this field should be determined by some process outside calibration, and changed only rarely.
- Channel Quality. This field can assume the values *Good, Acceptable,* or Bad.
- Channel Condition. This field can assume the values *OK, Noisy, Hot, Dead,* or *Other.*
- Fit status. This describes how well the fit performed when it produced this channel. This is field is not set during validation, but during fitting, by a CalFit object. This field can assume the values *FitOK*, *Unconverged*, *FitFailed*, and *FitInvalid*.

15.2 CalHeader flag fields

As with CalChan, the CalHeader flag fields are divided into public and private. The private fields of the CalHeader flag describe general attributes of the bank that owns it, while the public field can be defined as desired for different calibration types. A future release of the calibration system will allow searching the database for objects whose CalHeader flags satisfy an input (bit mask) condition.

The private CalHeader fields are defined in the following list. Most of these fields should be set prior to storing a CalBase object. The enums and bit fields for these are defined in **BbrCalib**/CalFlags.hh.

- CalBase Validity. This describes whether the CalBase object has passed the validity test. It can assume the values *Validated* or *NotValidated.* As with the CalChan validity flag, this field can only be set by the CalType base class.
- CalBase Reference. This defines whether the stored object should be used as a reference against which to compare new data before storing it. The content check of a CalBase is made against the most recent object with this bit set. It can assume the values *NotRefSet* or*RefSet*.
- CalBase Permanence. This defines how permanent the persistent store of this object should be. It can assume the values *Permanent, QuasiPermanent, Transitory,* or Volatile*.* Only permanent objects should be used in the configuration of the online system or the processing of event data. Once an object has been declared Permanent, it can never be removed from the database. Volatile objects are intended for short-term store only, much like a persistent scratch space. No code should depend in any way on volatile objects. The exact definition of QuasiPermanent and Transitory objects will be worked out in practice, perhaps differently for different types.
- CalBase OnlineUse. This defines where the calibration object will be used in the online system. The enum values correspond to unique bits in a bit field, as a single object may have multiple uses. Valid values are *None, Front End, ROM, ROC,* or*OEP.*
- CalBase OfflineUse. This defines what kind of offline processing (simulation, digi-making, hit reconstruction, etc.) use this data. As with OnlineUse, OfflineUse values correspond to unique bits to allow multiple values. The valid values for this have not yet been determined, though 16 bits have been reserved. Eventually it is intended to be able to select subsets of the database necessary for various levels of reprocessing from the value of this field.

15.3 CalBase functions

The CalBase function set is listed below. Those functions not yet implemented are marked with a. Garden-variety inspector and modifier functions are not listed here: people interested in seeing all the functions should look at **BbrCalib**/CalBase.hh.

♦ Constructors. These are only public through the fully implemented classes CalList, CalChanList, and CalIndirectList.
♦ CalBase(); // construct a dummy CalBase object, suitable only as a scratchpad for CalChans.
♦ CalBase(const char* sysname,const char* typename); // construct a CalBase which can store to or fetch from the database.
♦ CalBase(CalMapBrowser& ,const CalHeader&,const CalChan&,const char* typename); // Construct a CalBase using a map browser to define the system and channelID space.
♦ Unary functions
♦ void print(ostream&) const; // print a summary of the list.
♦ void printAll(ostream&) const; // print the contents of the list, one line per channel.
♦ void printHistory(ostream& output = cout) const; // print the history of the list
♦ † void statistics(ostream&) const; // Calculate and print mean, RMS and limits for all fields
♦ † int cut(bool select(CalChan*,void*),void*,bool anti=false); // Select CalChan objects for which the user-supplied 'select' function returns 'true'. If the 'anti' flag is set true, this returns the complementary set of CalChans. The return value indicates how many CalChan objects were removed.
♦ CalErr verify(BdbTime&, int objid)const ; //Verify the form and content against the most recent reference
♦ CalErr validate(BdbTime&, int objid); // validate the data as when storing. If this function succeeds, the *validate* bit will be set in the CalBase CalHeader, precluding repeating the validation test should the object be stored.
♦ CalErr store(BdbIntervalBase&, int objid, bool overridecontentcheck=false); //Store the CalBase as a particular object of a particular calibration type for the specified time. This calls-down to the next function, creating an interval which starts at the specified time and continues to +infinity.
♦ CalErr store(BdbIntervalBase&, int objid,bool overridecontentcheck=false); //Store the CalBase as a particular object of a particular calibration type for the specified time interval. See the CalType 'storeCalBase' function in appendix D for an explanation of the 'overridecontecheck' flag).
♦ CalErr fetch(BdbTime&, int objid);// Retrieve the specified calibration data from the database.
♦ int selectFlag(unsigned int flagval,unsigned int flagmask,bool anti=false); // Removes CalChans from the list whose flag values, when masked by the input mask, don't exactly equal the input value. If the 'anti' flag is set true, this returns the complementary set of CalChans. The return value indicates how many CalChan objects were removed.

Binary functions

- int selectID(const CalBase& other,bool anti=false); // Removes CalChans whose IDs are not present in the input CalBase object. If the 'anti' flag is set true, this creates the complementary set of CalChans. The input CalBase object need not have the same CalChan or CalHeader subclass. The return value indicates how many CalChan objects were removed.
- CalErr merge(CalBase&); //Merges the contents of the second list with the calling list. The history records of the input are copied with a higher index value. Only objects with the same CalChan and CalHeader subclasses can be merged.

15.4 CalType functions

The CalType class performs many functions, some of which are listed below. These functions are described only to give an understanding of the internal mechanisms involved, as the CalType interface should never be used directly. Only the 'validate' functions are described: exactly analogous 'verify' functions also exist, with nearly identical implementations. In addition to the functions described below, CalType has normal accessor functions that provide the names of the CalBank, CalChan, and CalHeader subclasses used for its data. CalType also has functions that return the name and unit name for each data field of the CalChan subclass it uses.

- CalErr storeCalBase(int objectID, const BdbIntervalBase& interval, CalBase& list, bool overridecontentcheck = false); // This function converts a CalBase object to a CalBank, and stores it in the database according to the time interval specified. Before being stored, the CalBase structure and content are validated using the 'validate' function described below. If the CalBase 'validate' function has been explicitly called, the CalBase object is stored without rerunning the test. IFF the structure test succeeds but the content check fails AND the 'overridecontentcheck' flag is true, the object will still be stored.
- CalErr validateCalBase(int objectID,const BdbTime&,CalBase&); // This function validates the structure and content of the CalBase prior to storage using the functions described below.
- virtual CalErr verifyStructure(int objectID,const CalBase& list); // this function tests that the CalBase contains the correct CalHeader and set of CalChan objects as part of validation. This function can be overwritten to include other structure tests, but in that case the CalType base class function should be explicitly called in that overwritten function.
- virtual int validateContent(int objectID,const BdbTime& time,CalBase& list) const; // This function tests the content of every CalChan in the CalBase as part of validation. The return value is the number of CalChan objects that fail. The overall content test is defined by comparing this with the maximum allowed number of failing channels for the specified object. The default implementation of this function runs the test described below on each

channel. If this function is overwritten, this single channel function described below can be ignored. This function will return fail (return the size of the CalBase) unless a *XxxStatusType* data object is present in the database.

- ♦ BdbStatus setMaxFailValidation(int iobj,int ival); // this functions sets the maximum number of channels which are allowed to fail validation for this type for the given objectID. This function should be called before storing the CalType for the changes to have any affect on CalBase validation. The return value indicates success or failure in using internal Objectivity calls.
- ♦ virtual bool validateChannel(CalChan& testchan,const CalHeader& testhead, const CalChan& refChan,const CalHeader& refhead, const CalChan& statchan) const; // This function defines a prototype for validating a CalChan by comparing it against a reference copy, subject to the global flag status. This function has a null implementation. If the validateContent flag is not overwritten, validateChannel must be overwritten.
- ♦ CalErr storeCalType(bool newtype = false); // This function stores a copy of the calling object in the conditions database, subject to the result of the validation function described below. The object copy is stored without tests IFF the 'newtype' flag is set true and there are no existing objects of this type currently in the database.
- ♦ CalErr verifyTypeStructure(const CalType&) const; // This function checks that the CalChan and CalHeader subclass and interpretation of the current object match those of the most recently stored object of the same type.
- ♦ virtual CalErr verifyTypeContent(const CalType&) const = 0; // This function tests the values of the CalType subclass specific data members. It is the responsibility of the subclass developer to provide a reasonable implementation.

15.5 DIRC examples

The **DrcCond** package provides an example of subsystem specific CalSysDfn and CalType subclasses. Currently this package fully implements three calibration types relevant for the DIRC, which use generic CalChan and CalBank subclasses. It also contains a few simple test programs that allow DIRC calibration data to be generated (with random values), stored, and retrieved.

The DrcSysDfn class is a subclass of CalSysDfn, which fully describes the dataflow tag space for the DIRC. Because the DIRC has a simple and uniform dataflow hierarchy, this class contains only a few native data members, which it uses to implement the CalSysDfn virtual functions. By default the main constructor for this class builds the IR2 configuration. The constructor parameter defaults can be overwritten to describe a simpler configuration (IE a test stand). DrcSysDfn divides up the calibration objects according to ROMs, and defines the calibration object IDs to be the same as the module field values in the detector tag.

The DrcStatusType class defines the DIRC status calibration type (general channel status type). This uses the generic CalStatusBank class to store its type's data. This class has data members that define bit masks which, when applied to the public CalChan fields, define the private field Quality and Condition values. Eventually this class will contain code to merge the status flags from different DIRC calibration types to define the general channel status.

The DrcPedType class defines the calibration type for the DIRC ADC readout pedestals. It uses the generic CalMSBank to store its type's data, assuming the pedestals can be measured as a simple mean of the response to null stimulus. DrcPedType data members define the cuts that are used in the verifyChannel function to define good channel data (min/max pedestal value, maximum difference/significance from the reference value). This class uses the DrcFlags.hh file, which defines a bit in the public CalChan flag field for each of the different data tests it performs during channel verification. DrcPedType also has several bit masks data members, which when applied to the public flag field define the channel Quality and whether it passes verification (the return value of the verifyChannel function).

A similar class DrcT0Type defines the calibration type for DIRC TDC readout time offsets. It uses the generic CalMSNBank to store its type's data. The additional data word in CalMSNChan (number of samples) versus CalMSChan is necessary since time offsets will be measured using the light flasher system, which randomly illuminates each phototube (channel), giving each a different number of samples. Data members in DrcT0Type define cuts on the channel data and masks for defining the flags similarly to DrcPedType.

The **DrcCond** package contains an implementation of CalVTblInit (DrcVTblInit) which forces virtual table loading for the calibration types used by the dirc. This class is linked to provide a dirc-specific calibration database browser executable, which is built by that package.

There are three test programs in the **DrcCond** package which fully exercise the system. StoreDrcTypes creates persistent objects for the three calibration types mentioned above. This program must be run first. StoreDrcData creates CalBase objects for the three calibration types listed above, and stores them. A complete set of calibration data objects (12 objects of roughly 10K channels) is created for each type, with random values in place of measurements. The StoreDrcType and StoreDrcData programs may be run repeatedly. The FetchDrcData program retrieves data from the database and prints it to the screen. These test programs use DrcVTblInit to force virtual table loading.

The **DrcOEP** package contains example OEP modules for both slow and fast calibration. The module DrcT0RomCalib is intended to be run in OEP as the Unix end of a ROM calibration. Currently, this module takes the CalHist objects accumulated in the ROM and converts them to Heptuple histograms for inspection, using the tools defined in section 11.2. The module DrcRomCalib performs the same simple calibration taking feature-extracted data from the ROM, and performing accumulation in OEP.

The **DrcOnline** package contains example CalIterator subclasses designed to work on dirc raw data. The class DrcTDCOutputIterator is a CalIterator working off of feature extracted data, while DrcInputTCIterator is a CalIterator for accessing the dirc raw front-end data.

The **DrcOdf** package contains an example implementation of a simple ROM-based calibration. This uses the classes DrcT0Config, DrcT0Begin, DrcT0End, and DrcT0L1Accept. This calibration uses only minor cycles, with DrcT0Config skipping directly to that level. The global function DrcT0SegTest constructs a DrcT0ConfigAction object and attaches it to the FSM.